DFT Definition and Properties

DFT

The discrete Fourier transform (DFT) is the primary transform used for numerical computation in digital signal processing. It is very widely used for spectrum analysis, fast convolution, and many other applications. The DFT transforms N discrete-time samples to the same number of discrete frequency samples, and is defined as
Equation:

$$X(k)=\sum_{n=0}^{N-1} x(n)e^{-\left(i\frac{2\pi nk}{N}\right)}$$

The DFT is widely used in part because it can be computed very efficiently using fast Fourier transform (FFT) algorithms.

IDFT

The inverse DFT (IDFT) transforms N discrete-frequency samples to the same number of discrete-time samples. The IDFT has a form very similar to the DFT,
Equation:

$$x(n)=\frac{1}{N}\sum_{k=0}^{N-1} X(k)e^{i\frac{2\pi nk}{N}}$$

and can thus also be computed efficiently using FFTs.

DFT and IDFT properties

Periodicity

Due to the N-sample periodicity of the complex exponential basis functions $e^{i\frac{2\pi nk}{N}}$ in the DFT and IDFT, the resulting transforms are also periodic with N samples.

$$X(k+N)=X(k)$$

$$x(n)=x(n+N)$$

Circular Shift

A shift in time corresponds to a phase shift that is linear in frequency. Because of the periodicity induced by the DFT and IDFT, the shift is **circular**, or modulo N samples.

$$x((n-m) \bmod N) \quad X(k)e^{-\left(i\frac{2\pi km}{N}\right)}$$

The modulus operator $p \bmod N$ means the **remainder** of p when divided by N. For example,

$$9 \bmod 5 = 4$$

and

$$-1 \bmod 5 = 4$$

Time Reversal

$$x((-n) \bmod N) = x((N-n) \bmod N) \quad X((N-k) \bmod N) = X((-k) \bmod N)$$

Note: time-reversal maps $0 \quad 0$, $1 \quad N-1$, $2 \quad N-2$, etc. as illustrated in the figure below.

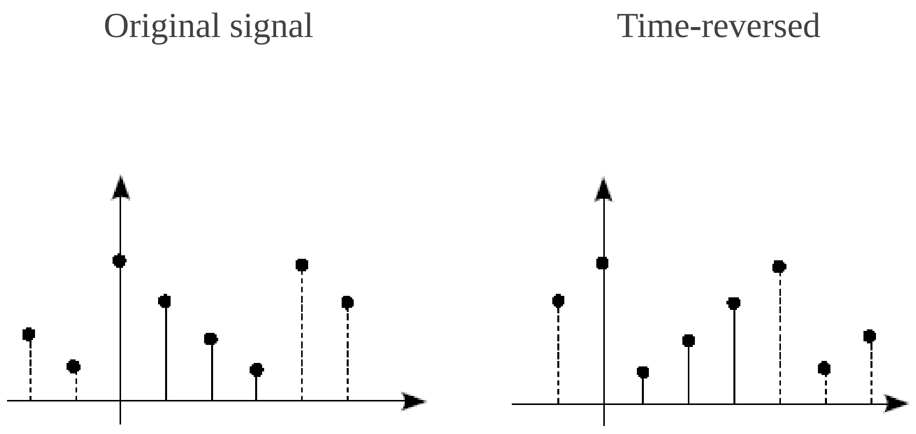


Illustration of circular time-reversal

Complex Conjugate

$$x(n) \quad X((-k) \bmod N)$$

Circular Convolution Property

Circular convolution is defined as

$$x(n)*h(n) \doteq \sum_{m=0}^{N-1} x(m)x((n-m) \bmod N)$$

Circular convolution of two discrete-time signals corresponds to multiplication of their DFTs:

$$x(n)*h(n) \quad X(k)H(k)$$

Multiplication Property

A similar property relates multiplication in time to circular convolution in frequency.

$$x(n)h(n) \quad \frac{1}{N}X(k)*H(k)$$

Parseval's Theorem

Parseval's theorem relates the energy of a length-N discrete-time signal (or one period) to the energy of its DFT.

$$\sum_{n=0}^{N-1} (|x(n)|)^2 = \frac{1}{N} \sum_{k=0}^{N-1} (|X(k)|)^2$$

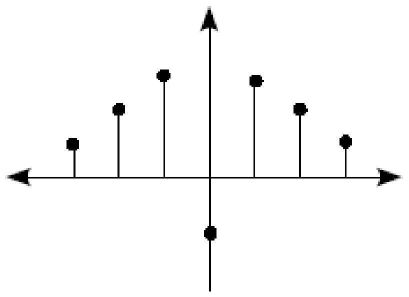
Symmetry

The continuous-time Fourier transform, the DTFT, and DFT are all defined as transforms of complex-valued data to complex-valued spectra. However, in practice signals are often real-valued. The DFT of a real-valued discrete-time signal has a special symmetry, in which the real part of the transform values are **DFT even symmetric** and the imaginary part is **DFT odd symmetric**, as illustrated in the equation and figure below.

$x(n)$ real $\quad X(k) = X((N-k) \bmod N)$ (This implies $X(0)$, $X\left(\frac{N}{2}\right)$ are real-valued.)

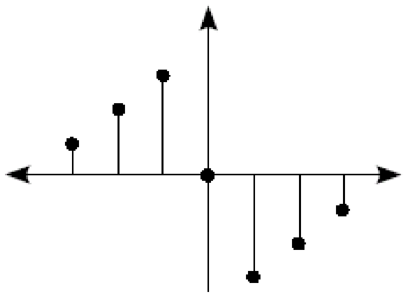
DFT symmetry of real-valued signal

Real part of X(k) is even



Even-symmetry in DFT sense

Imaginary part of X(k) is odd



Odd-symmetry in DFT sense

Spectrum Analysis Using the Discrete Fourier Transform

Discrete-Time Fourier Transform

The Discrete-Time Fourier Transform (DTFT) is the primary theoretical tool for understanding the frequency content of a discrete-time (sampled) signal. The DTFT is defined as
Equation:

$$X(\omega) = \sum_{n=-\infty}^{\infty} x(n) e^{-(i\omega n)}$$

The inverse DTFT (IDTFT) is defined by an integral formula, because it operates on a continuous-frequency DTFT spectrum:
Equation:

$$x(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} X(k) e^{i\omega n} \, \mathrm{d}\, \omega$$

The DTFT is very useful for theory and analysis, but is not practical for numerically computing a spectrum digitally, because

1. infinite time samples means
    - infinite computation
    - infinite delay
2. The transform is continuous in the discrete-time frequency, ω

For practical computation of the frequency content of real-world signals, the Discrete Fourier Transform (DFT) is used.

Discrete Fourier Transform

The DFT transforms N samples of a discrete-time signal to the same number of discrete frequency samples, and is defined as
Equation:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-\frac{i2\pi nk}{N}}$$

The DFT is invertible by the inverse discrete Fourier transform (IDFT):
Equation:

$$x(n) = \frac{1}{\mathrm{N}} \sum_{k=0}^{N-1} X(k) e^{i\frac{2\pi nk}{N}}$$

The DFT and IDFT are a self-contained, one-to-one transform pair for a length-N discrete-time signal. (That is, the DFT is not **merely** an approximation to the DTFT as discussed next.) However, the DFT is very often used as a practical approximation to the DTFT.

Relationships Between DFT and DTFT

DFT and Discrete Fourier Series

The DFT gives the discrete-time Fourier series coefficients of a periodic sequence ($x(n) = x(n+N)$) of period N samples, or
Equation:

$$X(\omega) = \frac{2\pi}{N} \sum X(k) \delta\left(\omega - \frac{2\pi k}{N}\right)$$

as can easily be confirmed by computing the inverse DTFT of the corresponding line spectrum:
Equation:

$$\begin{aligned} x(n) &= \frac{1}{2\pi} \int_{-\pi}^{\pi} \left(\frac{2\pi}{N} \sum X(k) \delta\left(\omega - \frac{2\pi k}{N}\right)\right) e^{i\omega n} \, \mathrm{d}\,\omega \\ &= \frac{1}{\mathrm{N}} \sum_{k=0}^{N-1} X(k) e^{i\frac{2\pi nk}{N}} \\ &= \mathrm{IDFT}\,(X(k)) \\ &= x(n) \end{aligned}$$

The DFT can thus be used to **exactly** compute the relative values of the N line spectral components of the DTFT of any periodic discrete-time sequence with an integer-length period.

DFT and DTFT of finite-length data

When a discrete-time sequence happens to equal zero for all samples except for those between *0* and $N - 1$, the infinite sum in the DTFT equation becomes the same as the finite sum from *0* to $N - 1$ in the DFT equation. By matching the arguments in the exponential terms, we observe that the DFT values **exactly** equal the DTFT for specific DTFT frequencies $\omega_k = \frac{2\pi k}{N}$. That is, the DFT computes exact samples of the DTFT at N equally spaced frequencies $\omega_k = \frac{2\pi k}{N}$, or

$$X\left(\omega_k = \frac{2\pi k}{N}\right) = \sum_{n=-\infty}^{\infty} x(n) e^{-(i\omega_k n)} = \sum_{n=0}^{N-1} x(n) e^{-\frac{i2\pi nk}{N}} = X(k)$$

DFT as a DTFT approximation

In most cases, the signal is neither exactly periodic nor truly of finite length; in such cases, the DFT of a finite block of N consecutive discrete-time samples does **not** exactly equal samples of the DTFT at specific frequencies. Instead, the DFT gives frequency samples of a windowed (truncated) DTFT

$$\hat{X}\left(\omega_k = \frac{2\pi k}{N}\right) = \sum_{n=0}^{N-1} x(n) e^{-(i\omega_k n)} = \sum_{n=-\infty}^{\infty} x(n) w(n) e^{-(i\omega_k n)} = X(k)$$

where $w(n) = \begin{cases} 1 & \text{if } 0 \leq n < N \\ 0 & \text{if else} \end{cases}$ Once again, $X(k)$ **exactly** equals $X(\omega_k)$ a DTFT frequency sample only when $\forall n, n \notin [0, N-1] : (x(n) = 0)$

Relationship between continuous-time FT and DFT

The goal of spectrum analysis is often to determine the frequency content of an analog (continuous-time) signal; very often, as in most modern spectrum analyzers, this is actually accomplished by sampling the analog signal, windowing (truncating) the data, and computing and plotting the magnitude of its DFT. It is thus essential to relate the DFT frequency samples back to the original analog frequency. Assuming that the analog signal is bandlimited and the sampling frequency exceeds twice that limit so that no frequency aliasing occurs, the relationship between the continuous-time Fourier frequency Ω (in radians) and the DTFT frequency ω imposed by sampling is $\omega = \Omega T$ where T is the sampling period. Through the relationship $\omega_k = \frac{2\pi k}{N}$ between the DTFT frequency ω and the DFT frequency index k, the correspondence between the DFT frequency index and the original analog frequency can be found:

$$\Omega = \frac{2\pi k}{NT}$$

or in terms of analog frequency f in Hertz (cycles per second rather than radians)

$$f = \frac{k}{NT}$$

for k in the range k between 0 and $\frac{N}{2}$. It is important to note that $k \in \left[\frac{N}{2}+1, N-1\right]$ correspond to **negative** frequencies due to the periodicity of the DTFT and the DFT.

Exercise:

Problem:

In general, will DFT frequency values $X(k)$ **exactly** equal samples of the analog Fourier transform X_a at the corresponding frequencies? That is, will $X(k) = X_a\left(\frac{2\pi k}{NT}\right)$?

Solution:

In general, **NO**. The DTFT exactly corresponds to the continuous-time Fourier transform only when the signal is bandlimited and sampled at more than twice its highest frequency. The DFT frequency values exactly correspond to frequency samples of the DTFT only when the discrete-time signal is time-limited. However, a bandlimited continuous-time signal cannot be time-limited, so in general these conditions cannot both be satisfied.

It can, however, be true for a small class of analog signals which are not time-limited but happen to exactly equal zero at all **sample times** outside of the interval $n \in [0, N-1]$. The sinc function with a bandwidth equal to the Nyquist frequency and centered at $t = 0$ is an example.

Zero-Padding

If more than N equally spaced frequency samples of a length-N signal are desired, they can easily be obtained by **zero-padding** the discrete-time signal and computing a DFT of the longer length. In particular, if LN DTFT samples are desired of a length-N sequence, one can compute the length-LN DFT of a length-LN zero-padded sequence

$$z(n) = \begin{cases} x(n) \text{ if } 0 \leq n \leq N-1 \\ 0 \text{ if } N \leq n \leq LN-1 \end{cases}$$

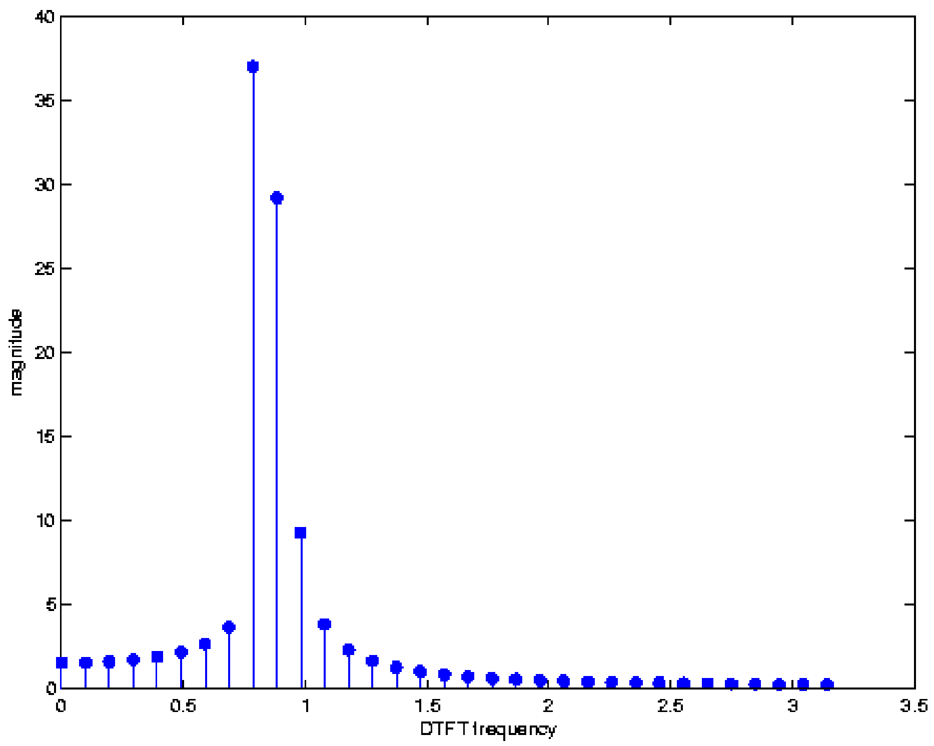
$$X\left(w_k = \frac{2\pi k}{LN}\right) = \sum_{n=0}^{N-1} x(n) e^{-\left(i\frac{2\pi kn}{LN}\right)} = \sum_{n=0}^{LN-1} z(n) e^{-\left(i\frac{2\pi kn}{LN}\right)} = \text{DFT}_{LN}\left[z[n]\right]$$

Note that zero-padding **interpolates** the spectrum. One should always zero-pad (by about at least a factor of 4) when using the DFT to approximate the DTFT to get a clear picture of the DTFT. While performing computations on zeros may at first seem inefficient, using FFT algorithms, which generally expect the same number of input and output samples, actually makes this approach very efficient.

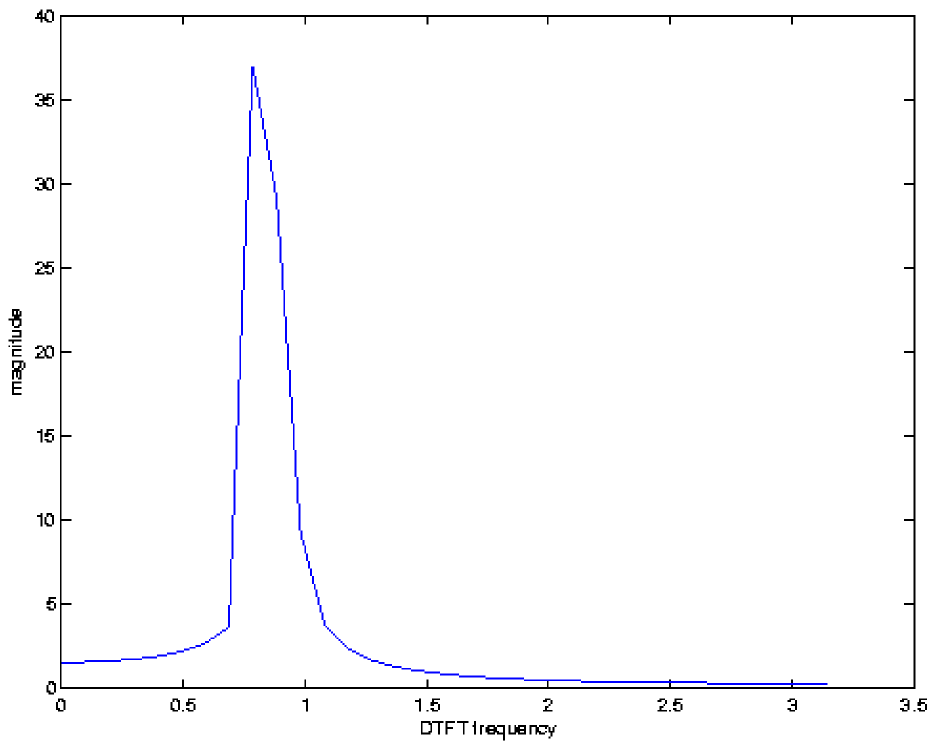
[link] shows the magnitude of the DFT values corresponding to the non-negative frequencies of a real-valued length-64 DFT of a length-64 signal, both in a "stem" format to emphasize the discrete nature of the DFT frequency samples, and as a line plot to emphasize its use as an approximation to the continuous-in-frequency DTFT. From this figure, it appears that the signal has a single dominant frequency component.
Spectrum without zero-padding

Magnitude DFT spectrum of 64 samples of a signal with a length-64 DFT
(no zero padding)

Stem plot



Line Plot

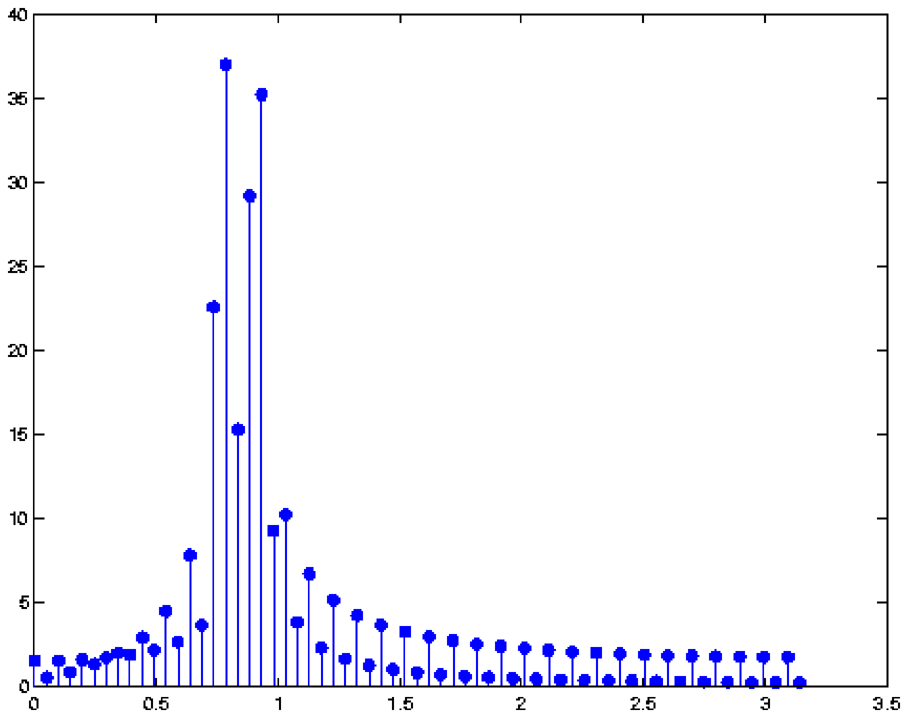


Zero-padding by a factor of two by appending 64 zero values to the signal and computing a length-128 DFT yields [link]. It can now be seen that the signal consists of at least two narrowband frequency components; the gap between them fell between DFT samples in [link], resulting in a misleading picture of the signal's spectral content. This is sometimes called the **picket-fence effect**, and is a result of insufficient sampling in frequency. While zero-padding by a factor of two has revealed more structure, it is unclear whether the peak magnitudes are reliably rendered, and the jagged linear interpolation in the line graph does not yet reflect the smooth, continuously-differentiable spectrum of the DTFT of a finite-length truncated signal. Errors in the apparent peak magnitude due to insufficient frequency sampling is sometimes referred to as **scalloping loss**.

Spectrum with factor-of-two zero-padding

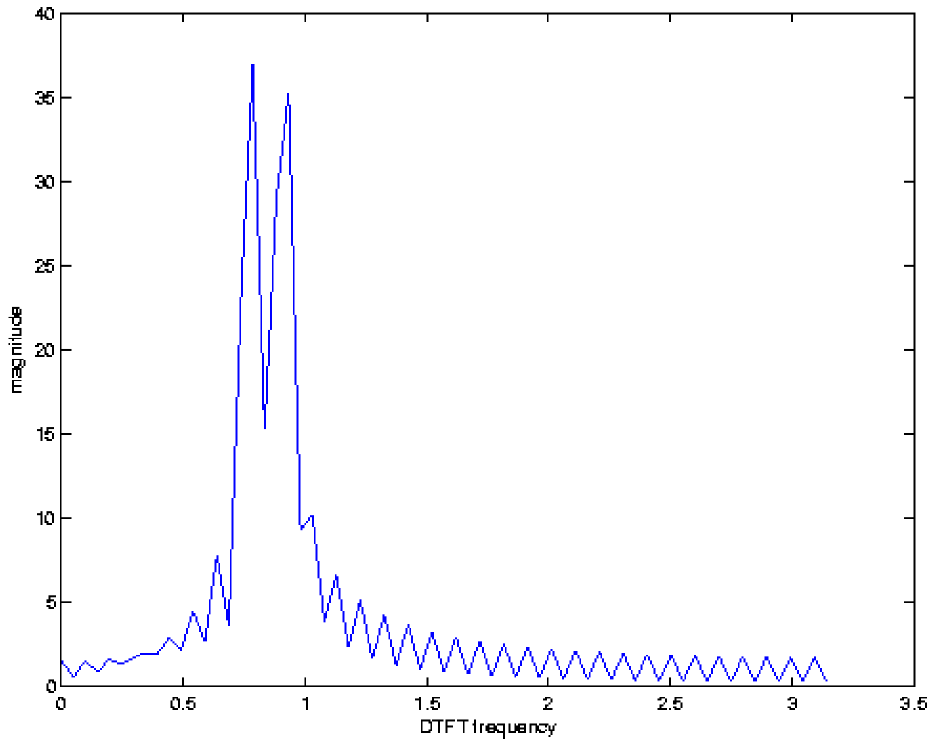
Magnitude DFT spectrum of 64 samples of a signal with a length-128 DFT (double-length zero-padding)

Stem plot



0
5
10
15
20
25
30
35
40
0
0.5
1
1.5
2
2.5
3
3.5

Line Plot



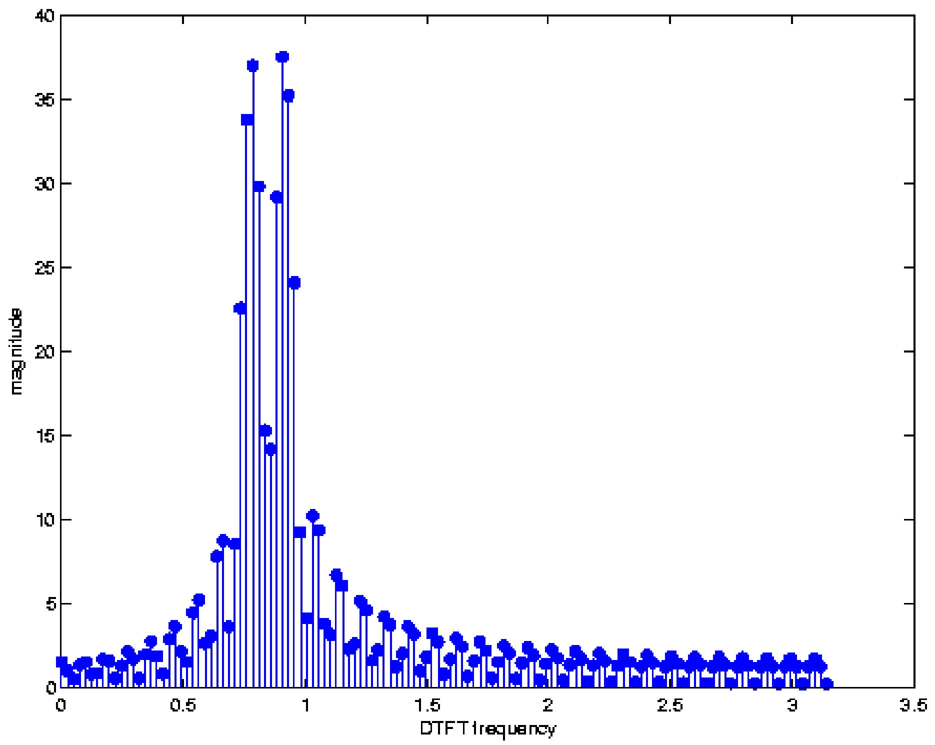
magnitude
DTFT frequency
0
5
10
15
20
25
30
35
40
0
0.5
1
1.5
2
2.5
3
3.5

Zero-padding to four times the length of the signal, as shown in [link], clearly shows the spectral structure and reveals that the magnitude of the two spectral lines are nearly identical. The line graph is still a bit rough and the peak magnitudes and frequencies may not be precisely captured, but the spectral characteristics of the truncated signal are now clear.

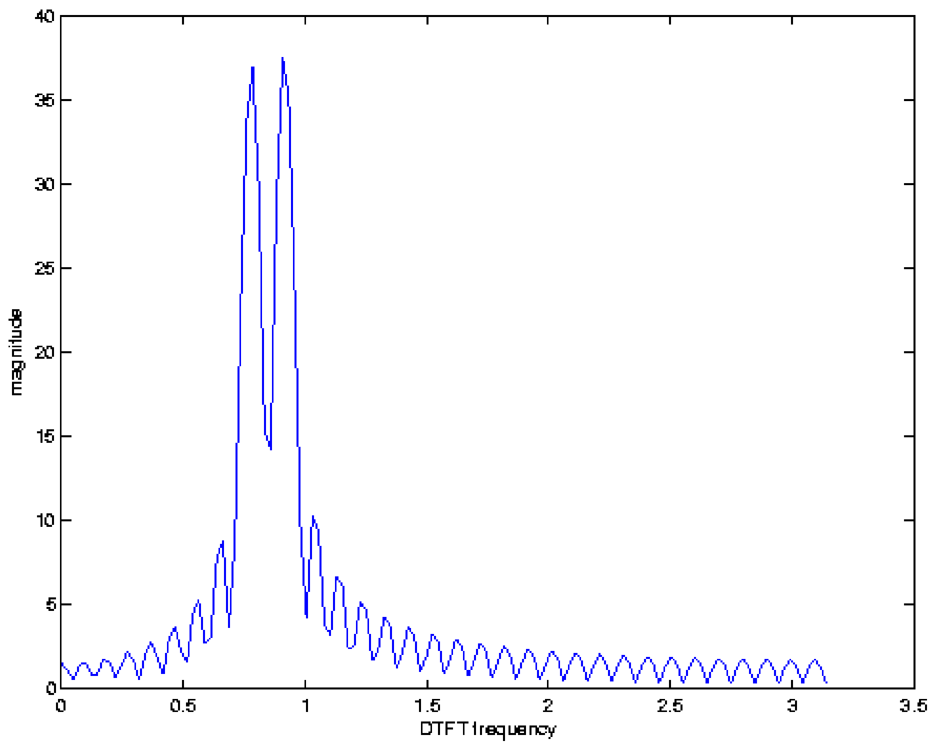
Spectrum with factor-of-four zero-padding

Magnitude DFT spectrum of 64 samples of a signal with a length-256 zero-padded DFT (four times zero-padding)

Stem plot



Line Plot

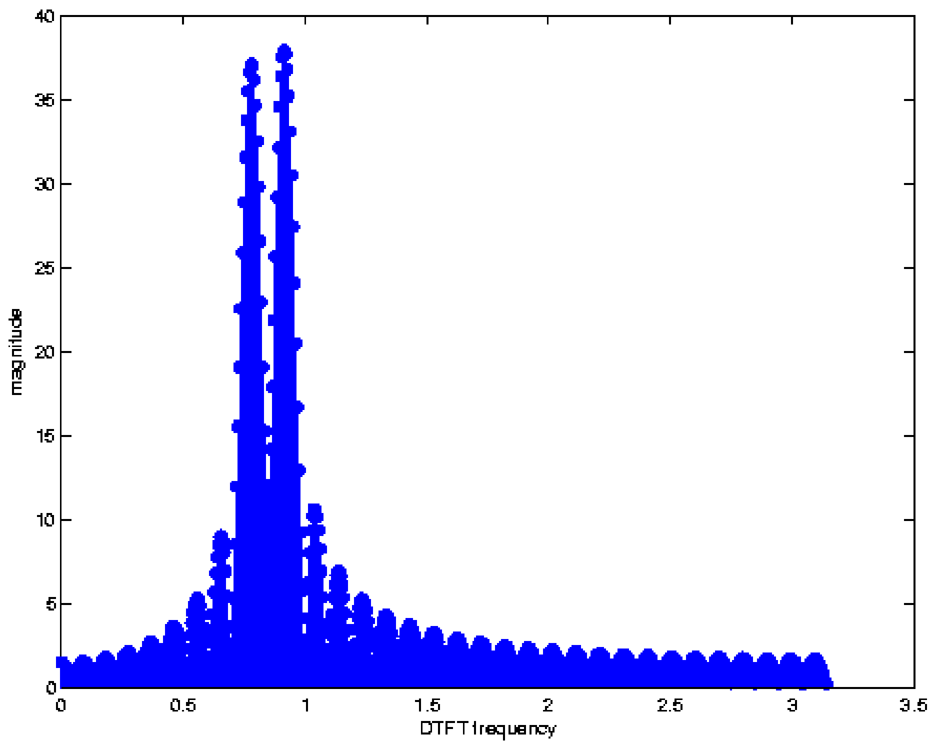


Zero-padding to a length of 1024, as shown in [link] yields a spectrum that is smooth and continuous to the resolution of the computer screen, and produces a very accurate rendition of the DTFT of the **truncated** signal.

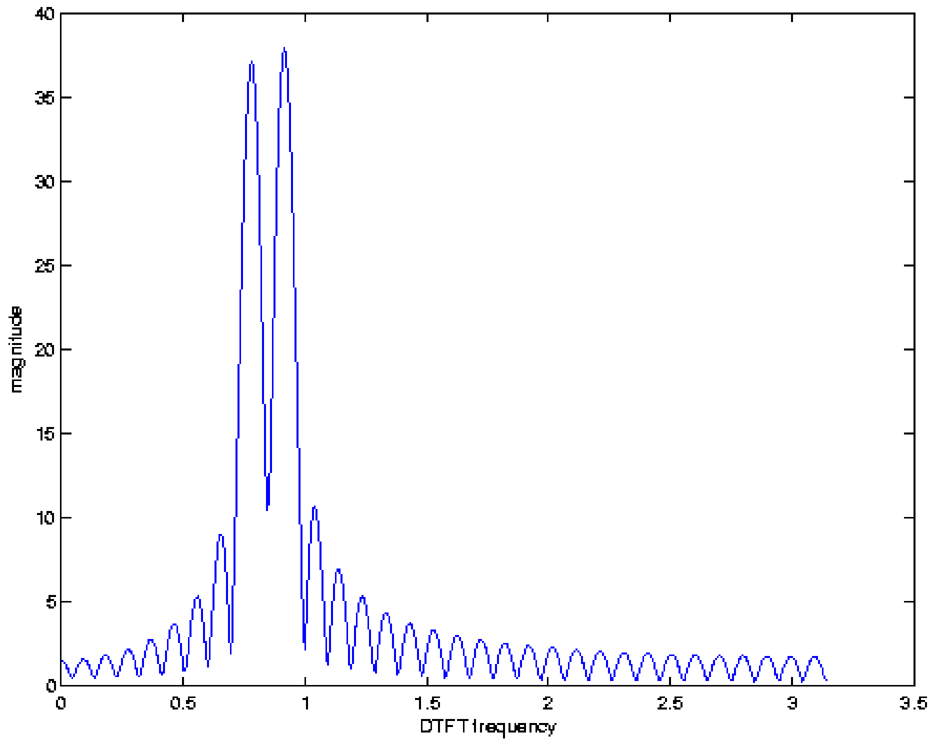
Spectrum with factor-of-sixteen zero-padding

Magnitude DFT spectrum of 64 samples of a signal with a length-1024 zero-padded DFT. The spectrum now looks smooth and continuous and reveals all the structure of the DTFT of a truncated signal.

Stem plot



Line Plot



The signal used in this example actually consisted of two pure sinusoids of equal magnitude. The slight difference in magnitude of the two dominant peaks, the breadth of the peaks, and the sinc-like lesser **side lobe** peaks throughout frequency are artifacts of the truncation, or windowing, process used to practically approximate the DFT. These problems and partial solutions to them are discussed in the following section.

Effects of Windowing

Applying the DTFT multiplication property

$$\widehat{X(\omega_k)} = \sum_{n=-\infty}^{\infty} x(n)w(n)e^{-(i\omega_k n)} = \frac{1}{2\pi}X(\omega_k)*W(\omega_k)$$

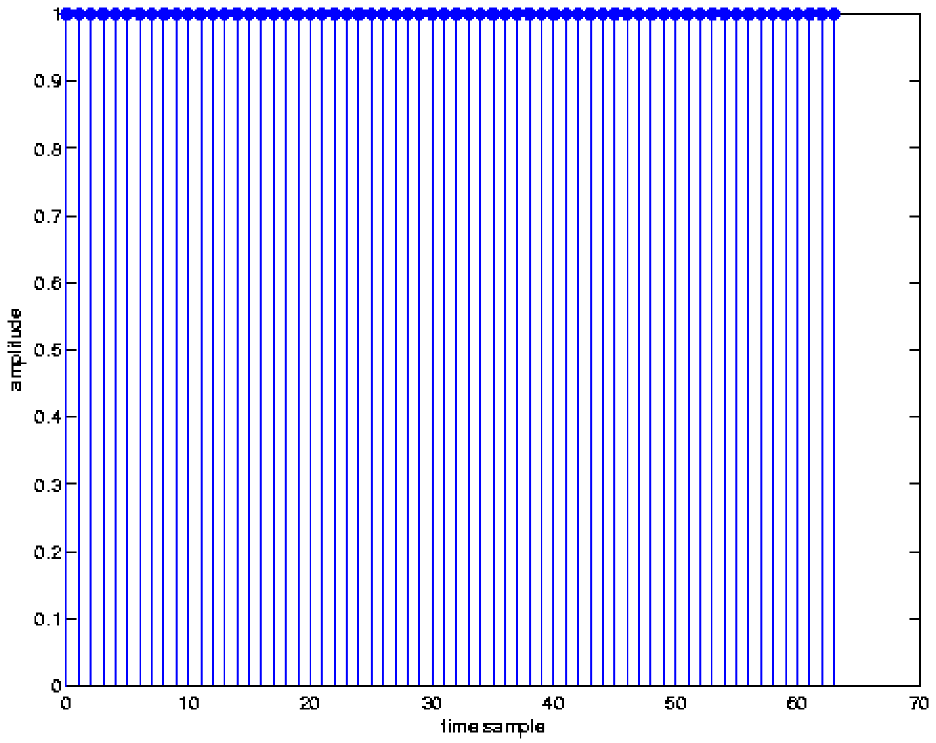
we find that the DFT of the windowed (truncated) signal produces samples not of the true (desired) DTFT spectrum $X(\omega)$, but of a **smoothed** verson $X(\omega)*W(\omega)$. We want this to resemble $X(\omega)$ as closely as possible, so $W(\omega)$ should be as close to an impulse as possible. The **window** $w(n)$ need not be a simple **truncation** (or **rectangle**, or **boxcar**) window; other shapes can also be used as long as they limit the sequence to at most N consecutive nonzero samples. All good windows are impulse-like, and represent various tradeoffs between three criteria:

1. main lobe width: (limits resolution of closely-spaced peaks of equal height)
2. height of first sidelobe: (limits ability to see a small peak near a big peak)
3. slope of sidelobe drop-off: (limits ability to see small peaks further away from a big peak)

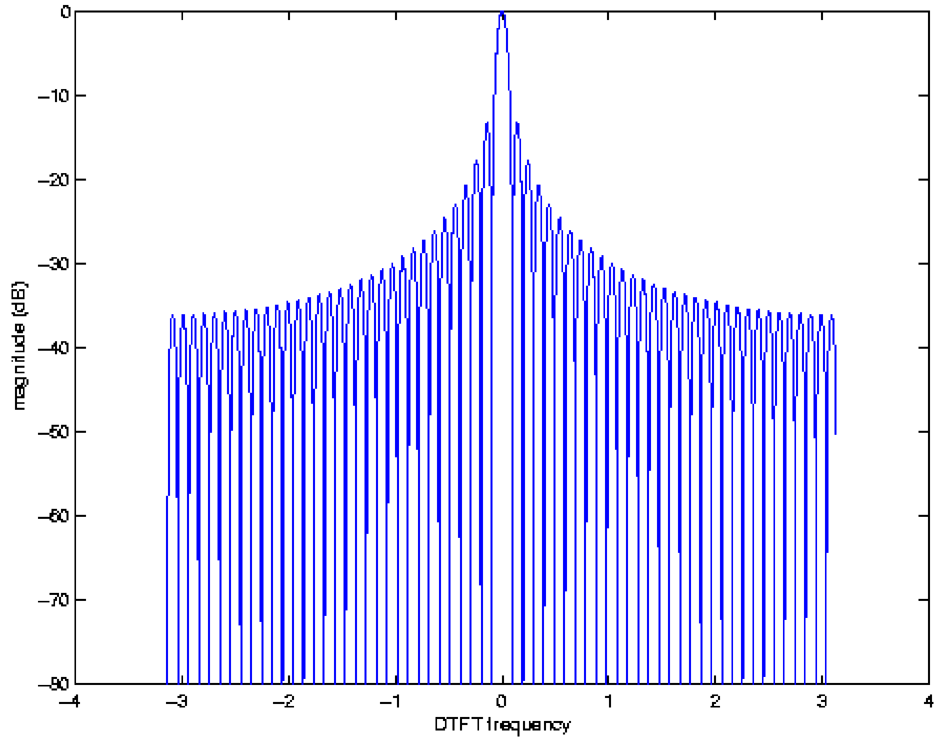
Many different window functions have been developed for truncating and shaping a length-N signal segment for spectral analysis. The simple **truncation** window has a periodic sinc DTFT, as shown in [link]. It has the narrowest main-lobe width, $\frac{2\pi}{N}$ at the -3 dB level and $\frac{4\pi}{N}$ between the two zeros surrounding the main lobe, of the common window functions, but also the largest side-lobe peak, at about -13 dB. The side-lobes also taper off relatively slowly.

Length-64 truncation (boxcar) window and its magnitude DFT spectrum

Rectangular window



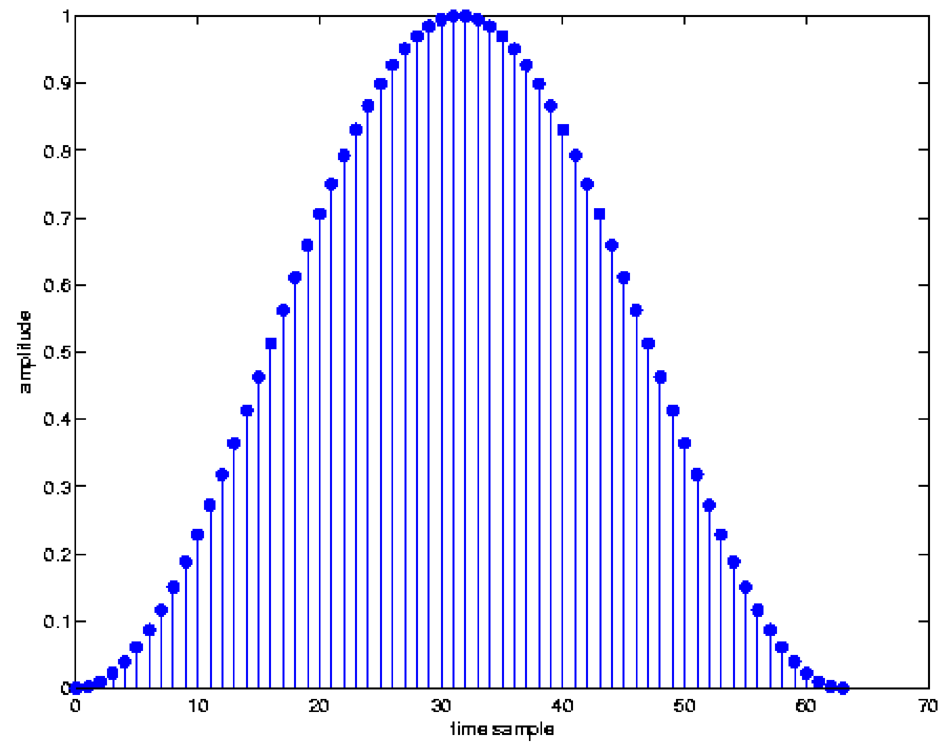
Magnitude of boxcar window spectrum



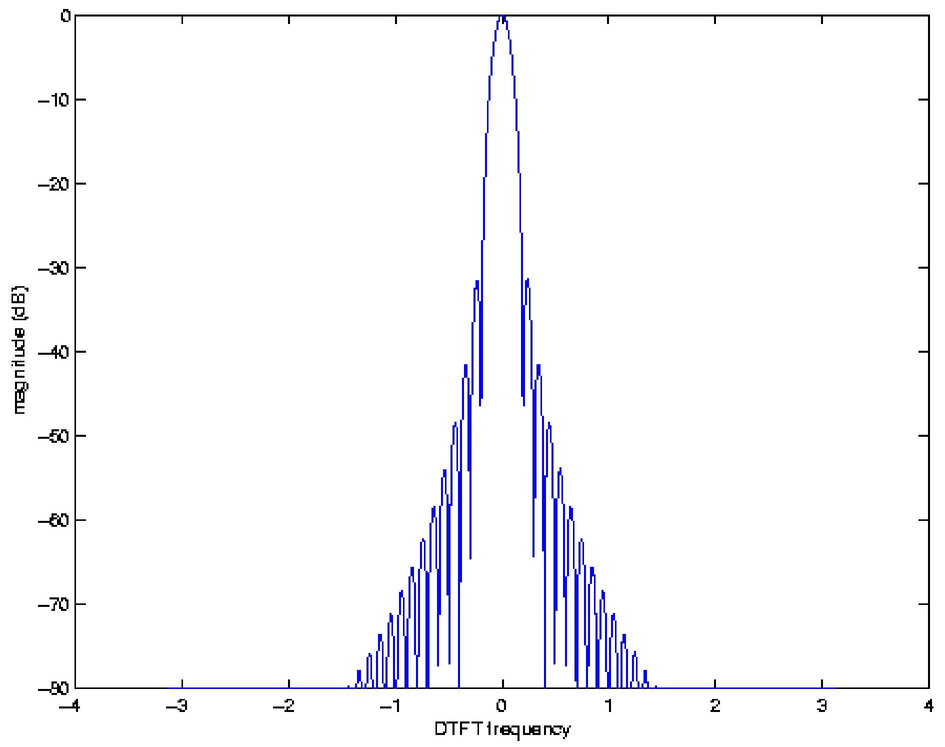
The **Hann window** (sometimes also called the **hanning** window), illustrated in [link], takes the form $w[n] = 0.5 - 0.5\cos\left(\frac{2\pi n}{N-1}\right)$ for n between 0 and $N-1$. It has a main-lobe width (about $\frac{3\pi}{N}$ at the -3 dB level and $\frac{8\pi}{N}$ between the two zeros surrounding the main lobe) considerably larger than the rectangular window, but the largest side-lobe peak is much lower, at about -31.5 dB. The side-lobes also taper off much faster. For a given length, this window is worse than the boxcar window at separating closely-spaced spectral components of similar magnitude, but better for identifying smaller-magnitude components at a greater distance from the larger components.

Length-64 Hann window and its magnitude DFT spectrum

Hann window



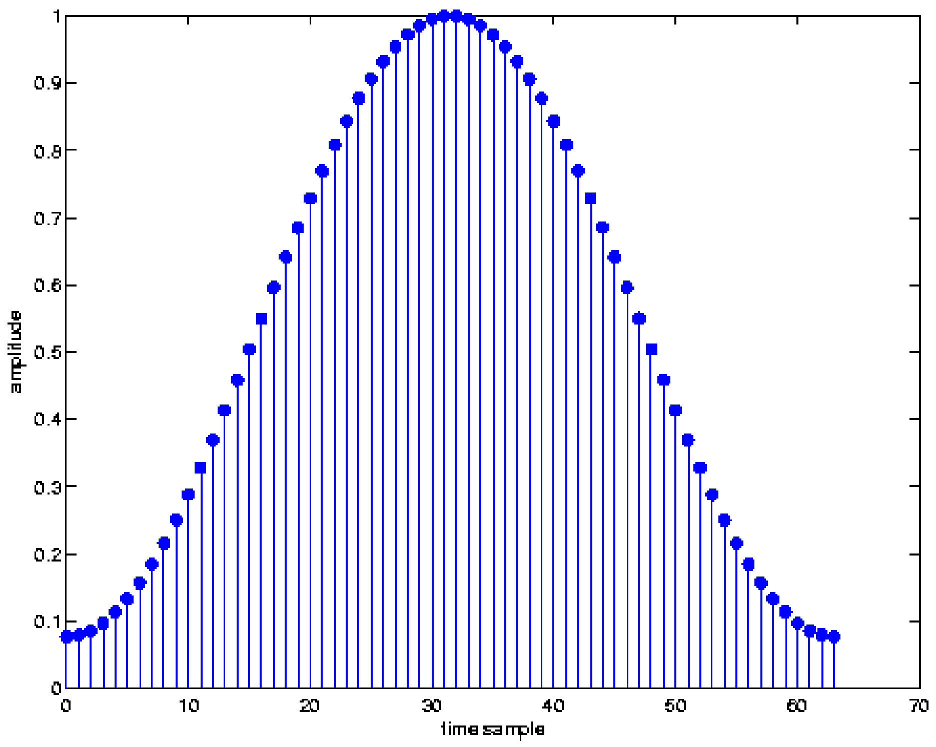
Magnitude of Hann window spectrum



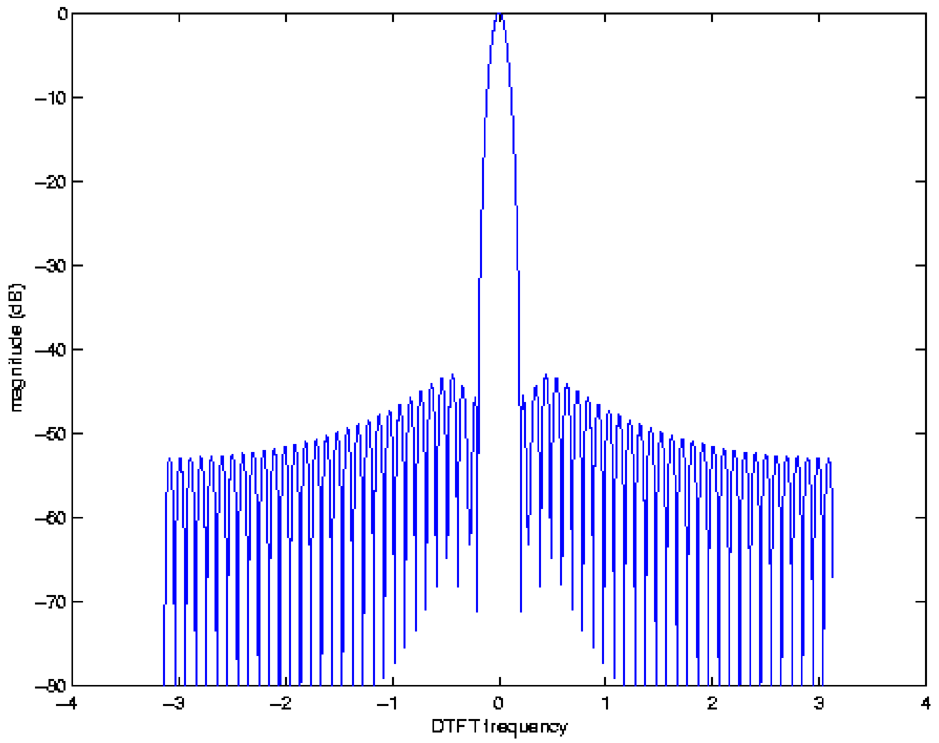
The **Hamming window**, illustrated in [link], has a form similar to the Hann window but with slightly different constants: $w[n] = 0.538 - 0.462\cos\left(\frac{2\pi n}{N-1}\right)$ for n between 0 and $N-1$. Since it is composed of the same Fourier series harmonics as the Hann window, it has a similar main-lobe width (a bit less than $\frac{3\pi}{N}$ at the -3 dB level and $\frac{8\pi}{N}$ between the two zeros surrounding the main lobe), but the largest side-lobe peak is much lower, at about -42.5 dB. However, the side-lobes also taper off much more slowly than with the Hann window. For a given length, the Hamming window is better than the Hann (and of course the boxcar) windows at separating a small component relatively near to a large component, but worse than the Hann for identifying very small components at considerable frequency separation. Due to their shape and form, the Hann and Hamming windows are also known as **raised-cosine windows**.

Length-64 Hamming window and its magnitude DFT spectrum

Hamming window



Magnitude of Hamming window spectrum

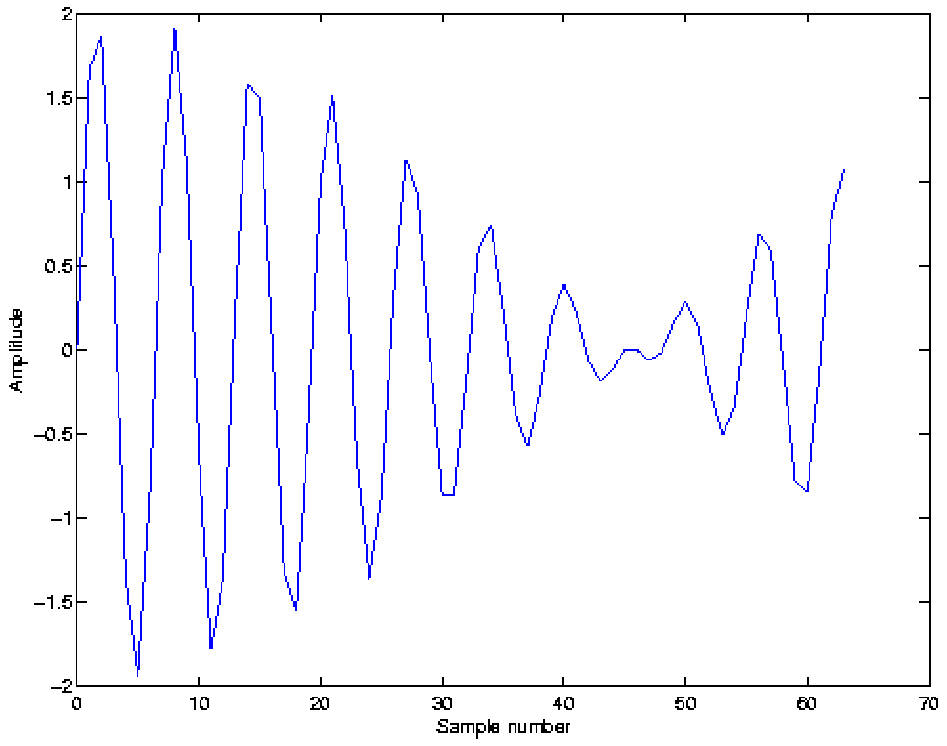


Note: Standard even-length windows are symmetric around a point halfway between the window samples $\frac{N}{2}-1$ and $\frac{N}{2}$. For some applications such as time-frequency analysis, it may be important to align the window perfectly to a sample. In such cases, a **DFT-symmetric** window that is symmetric around the $\frac{N}{2}$-th sample can be used. For example, the DFT-symmetric Hamming window is $w[n]=0.538-0.462\cos\left(\frac{2\pi n}{N}\right)$. A DFT-symmetric window has a purely real-valued DFT and DTFT. DFT-symmetric versions of windows, such as the Hamming and Hann windows, composed of few discrete Fourier series terms of period N, have few non-zero DFT terms (only when **not** zero-padded) and can be used efficiently in running FFTs.

The main-lobe width of a window is an inverse function of the window-length N; for any type of window, a longer window will always provide better resolution.

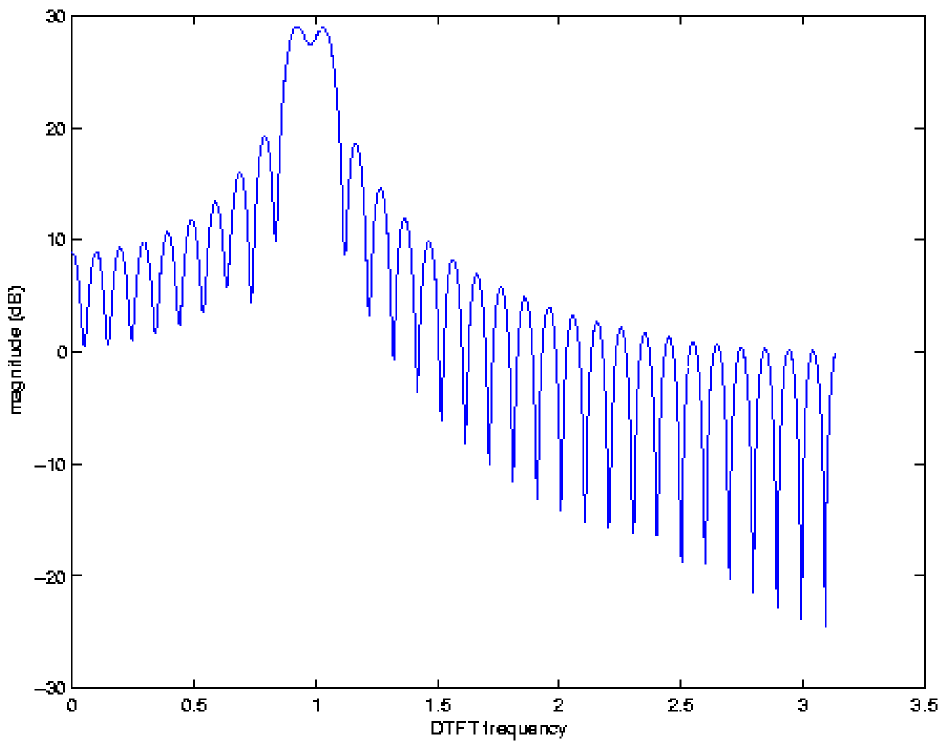
Many other windows exist that make various other tradeoffs between main-lobe width, height of largest side-lobe, and side-lobe rolloff rate. The Kaiser window family, based on a modified Bessel function, has an adjustable parameter that allows the user to tune the tradeoff over a continuous range. The Kaiser window has near-optimal time-frequency resolution and is widely used. A list of many different windows can be found here.

Example:
[link] shows 64 samples of a real-valued signal composed of several sinusoids of various frequencies and amplitudes.



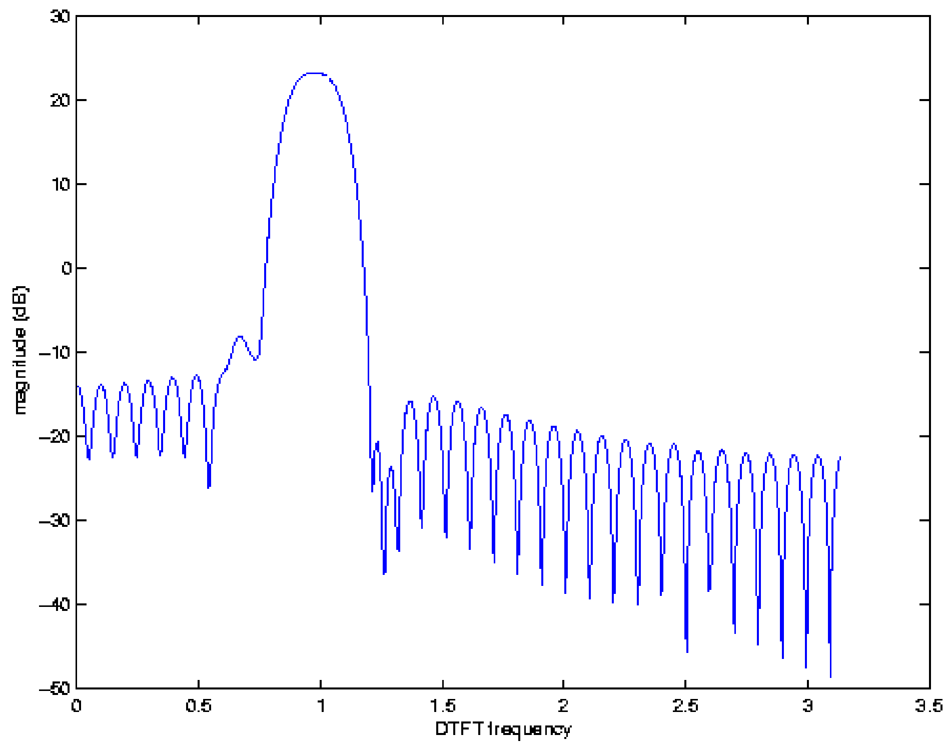
64 samples of an unknown signal

[link] shows the magnitude (in dB) of the positive frequencies of a length-1024 zero-padded DFT of this signal (that is, using a simple truncation, or rectangular, window).



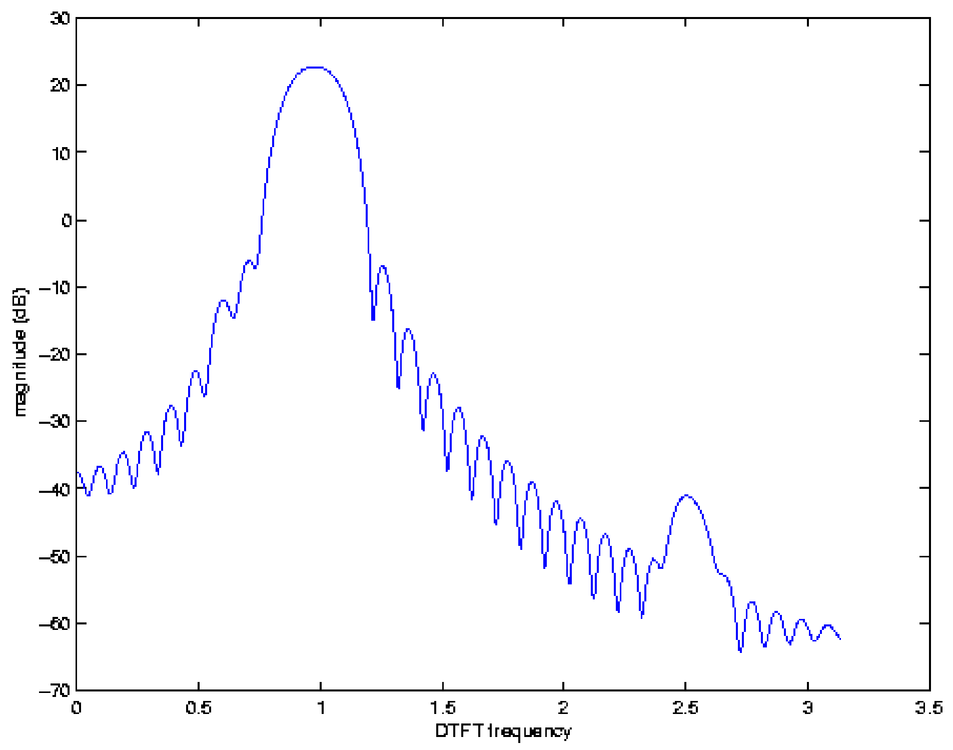
Magnitude (in dB) of the zero-padded DFT spectrum of the signal in [link] using a simple length-64 rectangular window

From this spectrum, it is clear that the signal has two large, nearby frequency components with frequencies near 1 radian of essentially the same magnitude. [link] shows the spectral estimate produced using a length-64 Hamming window applied to the same signal shown in [link].



Magnitude (in dB) of the zero-padded DFT spectrum of the signal in [link] using a length-64 Hamming window

The two large spectral peaks can no longer be resolved; they blur into a single broad peak due to the reduced spectral resolution of the broader main lobe of the Hamming window. However, the lower side-lobes reveal a third component at a frequency of about 0.7 radians at about 35 dB lower magnitude than the larger components. This component was entirely buried under the side-lobes when the rectangular window was used, but now stands out well above the much lower nearby side-lobes of the Hamming window. [link] shows the spectral estimate produced using a length-64 Hann window applied to the same signal shown in [link].



Magnitude (in dB) of the zero-padded DFT spectrum of the signal in [link] using a length-64 Hann window

The two large components again merge into a single peak, and the smaller component observed with the Hamming window is largely lost under the higher nearby side-lobes of the Hann window. However, due to the much faster side-lobe rolloff of the Hann window's spectrum, a fourth component at a frequency of about 2.5 radians with a magnitude about 65 dB below that of the main peaks is now clearly visible.

This example illustrates that no single window is best for all spectrum analyses. The best window depends on the nature of the signal, and different windows may be better for different components of the same signal. A skilled spectrum analysist may apply several different windows to a signal to gain a fuller understanding of the data.

Classical Statistical Spectral Estimation

Many signals are either partly or wholly stochastic, or random. Important examples include human speech, vibration in machines, and CDMA communication signals. Given the ever-present noise in electronic systems, it can be argued that almost **all** signals are at least partly stochastic. Such signals may have a distinct **average** spectral structure that reveals important information (such as for speech recognition or early detection of damage in machinery). Spectrum analysis of any single block of data using window-based deterministic spectrum analysis, however, produces a random spectrum that may be difficult to interpret. For such situations, the classical statistical spectrum estimation methods described in this module can be used.

The goal in classical statistical spectrum analysis is to estimate $E\left[(|X(\omega)|)^2\right]$, the **power spectral density (PSD)** across frequency of the stochastic signal. That is, the goal is to find the expected (mean, or average) energy density of the signal as a function of frequency. (For zero-mean signals, this equals the variance of each frequency sample.) Since the spectrum of each block of signal samples is itself random, we must average the squared spectral magnitudes over a number of blocks of data to find the mean. There are two main classical approaches, the periodogram and auto-correlation methods.

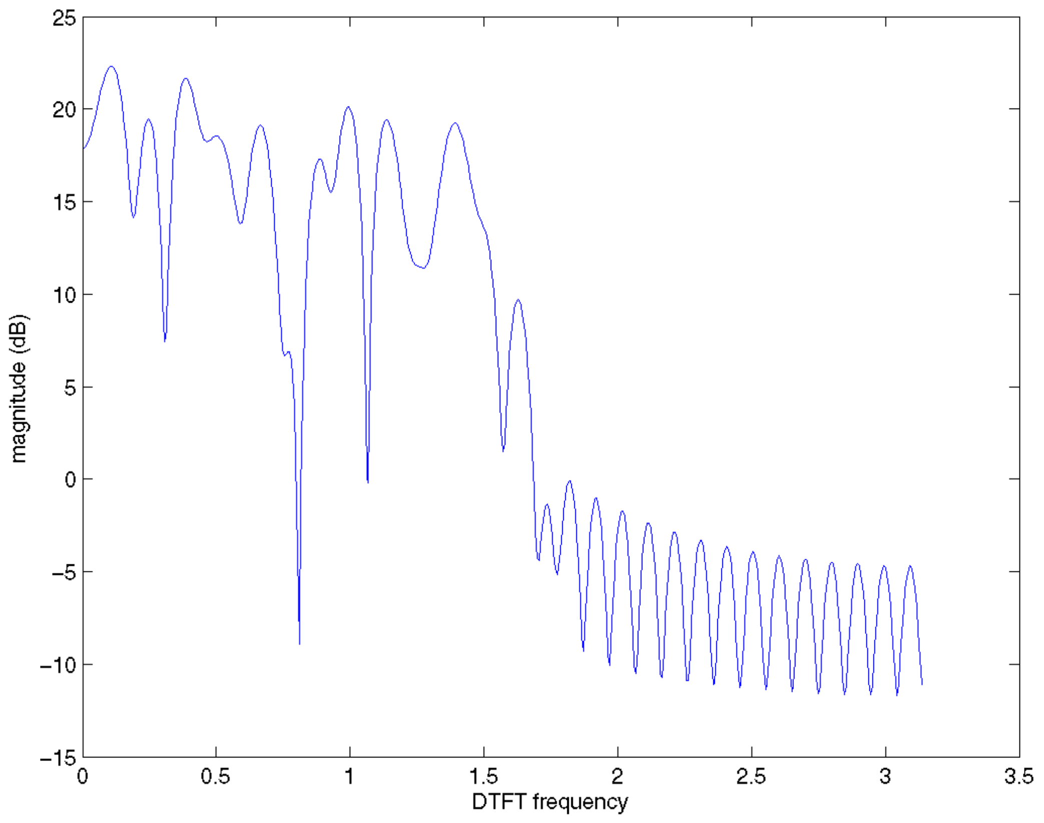
Periodogram method

The periodogram method divides the signal into a number of shorter (and often overlapped) blocks of data, computes the squared magnitude of the windowed (and usually zero-padded) DFT, $X_i(\omega_k)$, of each block, and averages them to estimate the power spectral density. The squared magnitudes of the DFTs of L possibly overlapped length-N windowed blocks of signal (each probably with zero-padding) are averaged to estimate the power spectral density:

$$\widehat{X(\omega_k)} = \frac{1}{L}\sum_{i=1}^{L}(|X_i(\omega_k)|)^2$$

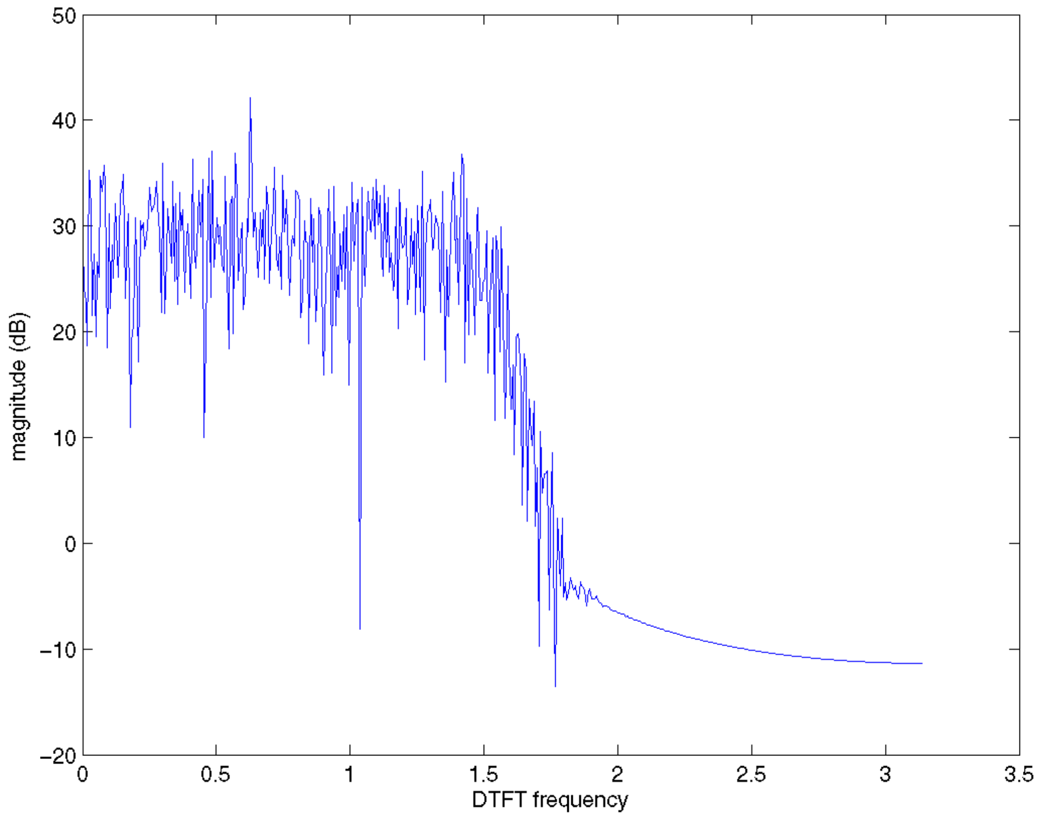
For a fixed **total** number of samples, this introduces a tradeoff: Larger individual data blocks provides better frequency resolution due to the use of a longer window, but it means there are less blocks to average, so the estimate has higher variance and appears more noisy. The best tradeoff depends on the application. Overlapping blocks by a factor of two to four increases the number of averages and reduces the variance, but since the same data is being reused, still more overlapping does not further reduce the variance. As with any window-based spectrum estimation procedure, the window function introduces broadening and sidelobes into the power spectrum estimate. That is, the periodogram produces an estimate of the **windowed** spectrum $\widehat{X(\omega)} = E\left[(|X(\omega)*W_M|)^2\right]$, not of $E\left[(|X(\omega)|)^2\right]$.

Example:
[link] shows the non-negative frequencies of the DFT (zero-padded to 1024 total samples) of 64 samples of a real-valued stochastic signal.



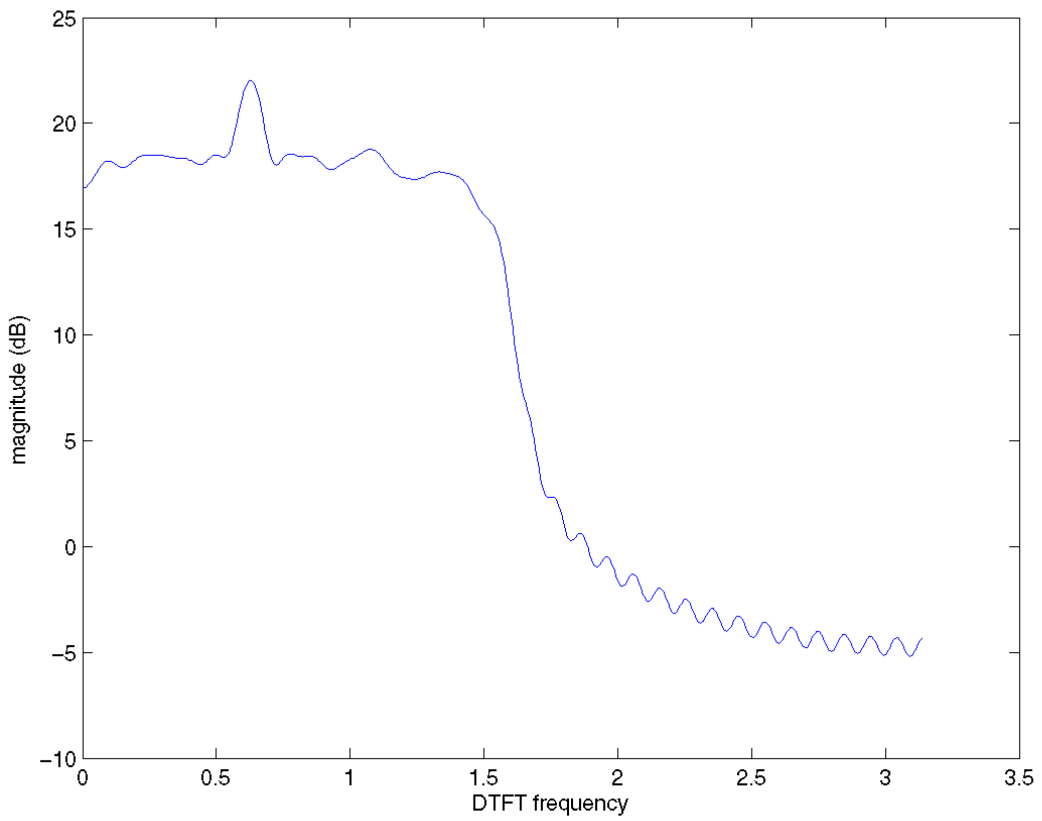
DFT magnitude (in dB) of 64 samples of a stochastic signal

With no averaging, the power spectrum is very noisy and difficult to interpret other than noting a significant reduction in spectral energy above about half the Nyquist frequency. Various peaks and valleys appear in the lower frequencies, but it is impossible to say from this figure whether they represent actual structure in the power spectral density (PSD) or simply random variation in this single realization. [link] shows the same frequencies of a length-1024 DFT of a length-1024 signal. While the frequency resolution has improved, there is still no averaging, so it remains difficult to understand the power spectral density of this signal. Certain small peaks in frequency might represent narrowband components in the spectrum, or may just be random noise peaks.



DFT magnitude (in dB) of 1024 samples of a stochastic signal

In [link], a power spectral density computed from averaging the squared magnitudes of length-1024 zero-padded DFTs of 508 length-64 blocks of data (overlapped by a factor of four, or a 16-sample step between blocks) are shown.



Power spectrum density estimate (in dB) of 1024 samples of a stochastic signal

While the frequency resolution corresponds to that of a length-64 truncation window, the averaging greatly reduces the variance of the spectral estimate and allows the user to reliably conclude that the signal consists of lowpass broadband noise with a flat power spectrum up to half the Nyquist frequency, with a stronger narrowband frequency component at around 0.65 radians.

Auto-correlation-based approach

The averaging necessary to estimate a power spectral density can be performed in the discrete-time domain, rather than in frequency, using the auto-correlation method. The squared magnitude of the frequency response, from the DTFT multiplication and conjugation properties, corresponds in the discrete-time domain to the signal convolved with the time-reverse of itself,

$$(|X(\omega)|)^2 = X(\omega)X^*(\omega) \leftrightarrow \left(x(n), x^*(-n)\right) = r(n)$$

or its **auto-correlation**

$$r(n) = \sum_k x(k)x^*(n+k)$$

We can thus compute the squared magnitude of the spectrum of a signal by computing the DFT of its auto-correlation. For stochastic signals, the power spectral density is an expectation, or average, and by linearity of expectation can be found by transforming the average of the auto-correlation. For a finite block of N signal samples, the average of the autocorrelation values, $r(n)$, is

$$r(n) = \frac{1}{N-n} \sum_{k=0}^{N-(1-n)} x(k)x^*(n+k)$$

Note that with increasing **lag**, n, fewer values are averaged, so they introduce more noise into the estimated power spectrum. By windowing the auto-correlation before transforming it to the frequency domain, a less noisy power spectrum is obtained, at the expense of less resolution. The multiplication property of the DTFT shows that the windowing smooths the resulting power spectrum via convolution with the DTFT of the window:

$$\widehat{X(\omega)} = \sum_{n=-M}^{M} r(n)w(n)e^{-(i\omega n)} = \left(E\left[(|X(\omega)|)^2\right]\right) * W(\omega)$$

This yields another important interpretation of how the auto-correlation method works: it estimates the power spectral density by **averaging the power spectrum over nearby frequencies**, through convolution with the window function's transform, to reduce variance. Just as with the periodogram approach, there is always a variance vs. resolution tradeoff. The periodogram and the auto-correlation method give similar results for a similar amount of averaging; the user should simply note that in the periodogram case, the window introduces smoothing of the spectrum via frequency convolution **before** squaring the magnitude, whereas the periodogram convolves the squared magnitude with $W(\omega)$.

Short Time Fourier Transform

Short Time Fourier Transform

The Fourier transforms (FT, DTFT, DFT, etc.) do not clearly indicate how the frequency content of a signal changes over time.

That information is hidden in the phase - it is not revealed by the plot of the magnitude of the spectrum.

Note: To see how the frequency content of a signal changes over time, we can cut the signal into blocks and compute the spectrum of each block.

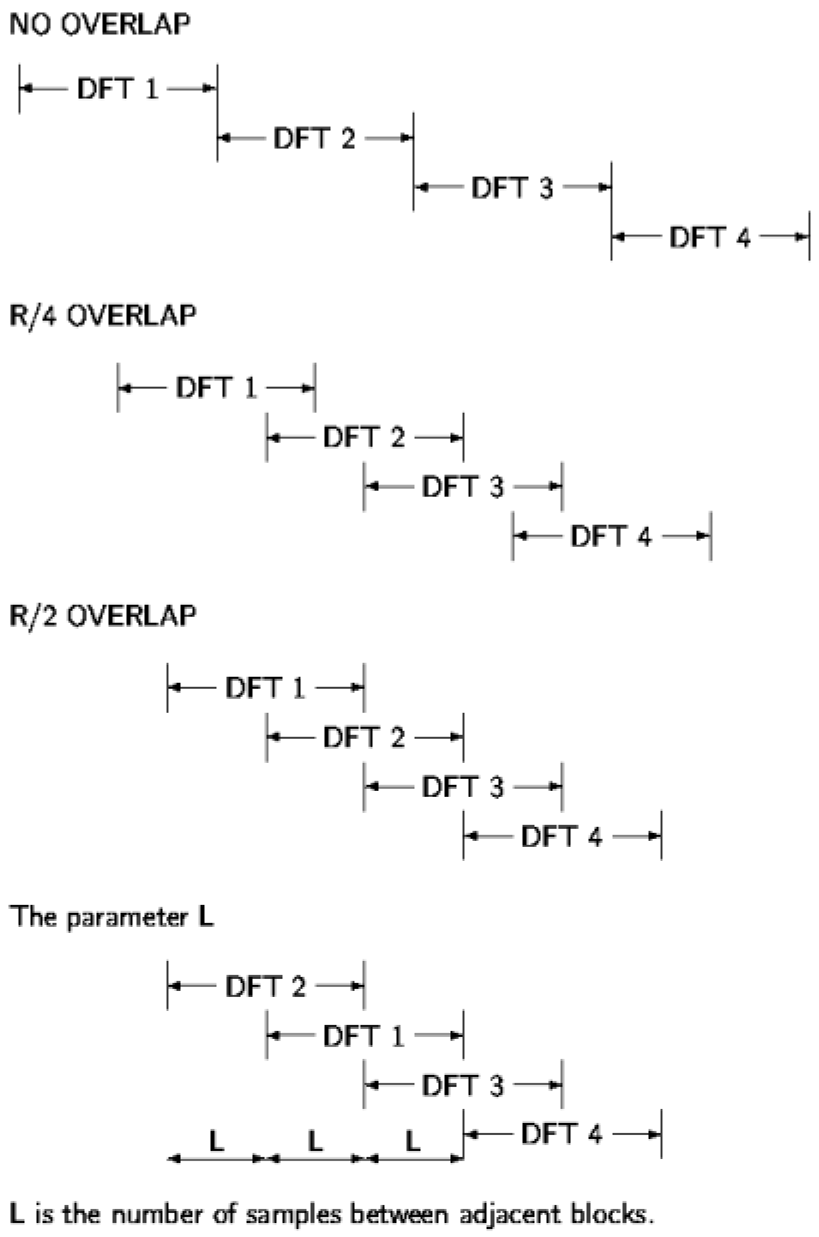
To improve the result,

1. blocks are overlapping
2. each block is multiplied by a window that is tapered at its endpoints.

Several parameters must be chosen:

- Block length, R.
- The type of window.
- Amount of overlap between blocks. ([link])
- Amount of zero padding, if any.

STFT: Overlap Parameter



The short-time Fourier transform is defined as
Equation:

$$\begin{aligned} X(\omega, m) &= \mathrm{STFT}\,(x(n)) \coloneqq \mathrm{DTFT}\,(x(n-m)w(n)) \\ &= \sum_{n=-\infty}^{\infty} x(n-m)w(n)e^{-(i\omega n)} \\ &= \sum_{n=0}^{R-1} x(n-m)w(n)e^{-(i\omega n)} \end{aligned}$$

where $w(n)$ is the window function of length R.

1. The STFT of a signal $x(n)$ is a function of two variables: time and frequency.
2. The block length is determined by the support of the window function $w(n)$.
3. A graphical display of the magnitude of the STFT, $|X(\omega, m)|$, is called the **spectrogram** of the signal. It is often used in speech processing.
4. The STFT of a signal is invertible.
5. One can choose the block length. A long block length will provide higher frequency resolution (because the main-lobe of the window function will be narrow). A short block length will provide higher time resolution because less averaging across samples is performed for each STFT value.
6. A **narrow-band spectrogram** is one computed using a relatively long block length R, (long window function).
7. A **wide-band spectrogram** is one computed using a relatively short block length R, (short window function).

Sampled STFT

To numerically evaluate the STFT, we sample the frequency axis ω in N equally spaced samples from $\omega = 0$ to $\omega = 2\pi$.
Equation:

$$\forall k, 0 \leq k \leq N-1 : \left(\omega_k = \frac{2\pi}{N} k\right)$$

We then have the discrete STFT,
Equation:

$$\begin{array}{rcl} X^d(k,m) \coloneqq X\left(\frac{2\pi}{N}k, m\right) & = & \sum_{n=0}^{R-1} x(n-m)w(n)e^{-(i\omega n)} \\ & = & \sum_{n=0}^{R-1} x(n-m)w(n){W_N}^{-(kn)} \\ & = & \mathrm{DFT}_N\left(x(n-m)w(n)|_{n=0}^{R-1}, 0, \ldots 0\right) \end{array}$$

where 0,. . .0 is $N - R$.

In this definition, the overlap between adjacent blocks is $R - 1$. The signal is shifted along the window one sample at a time. That generates more points than is usually needed, so we also sample the STFT along the time direction. That means we usually evaluate

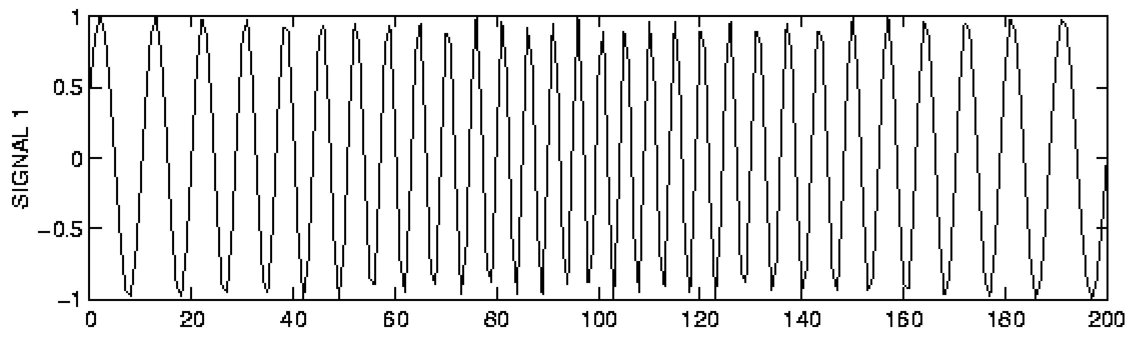
$$X^d(k, Lm)$$

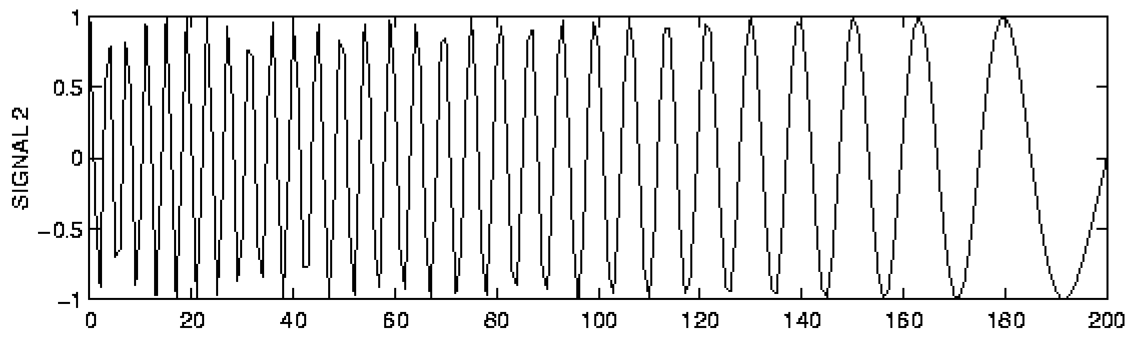
where L is the time-skip. The relation between the time-skip, the number of overlapping samples, and the block length is

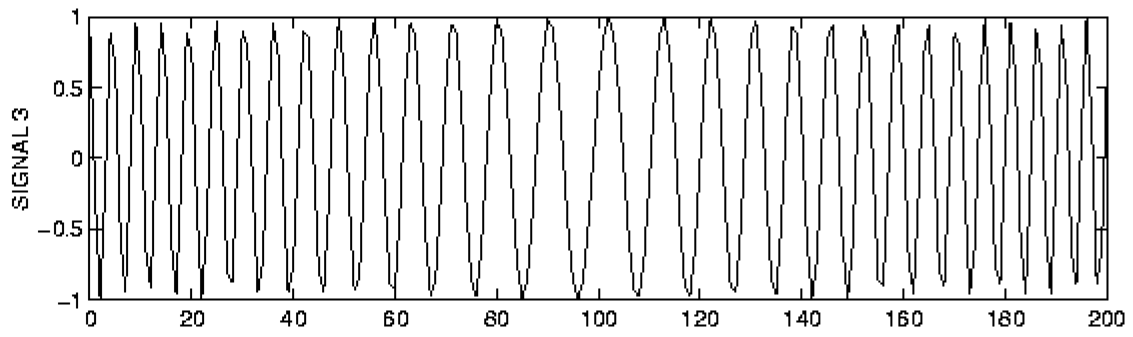
$$\text{Overlap} = R - L$$

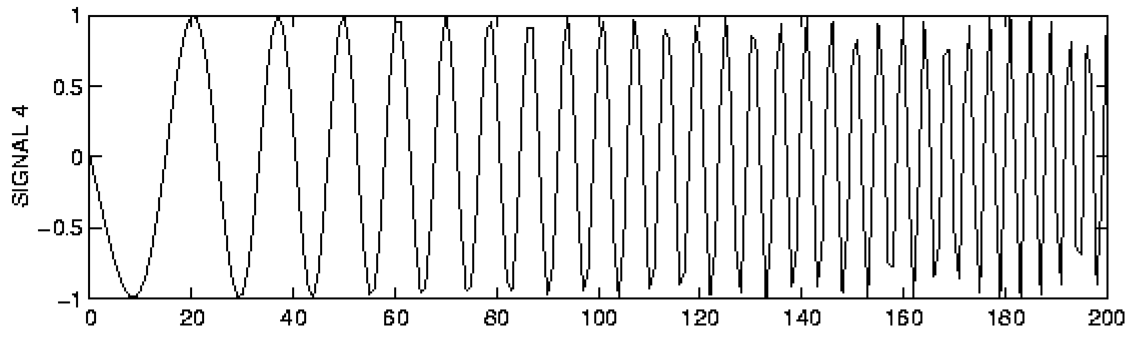
Exercise:

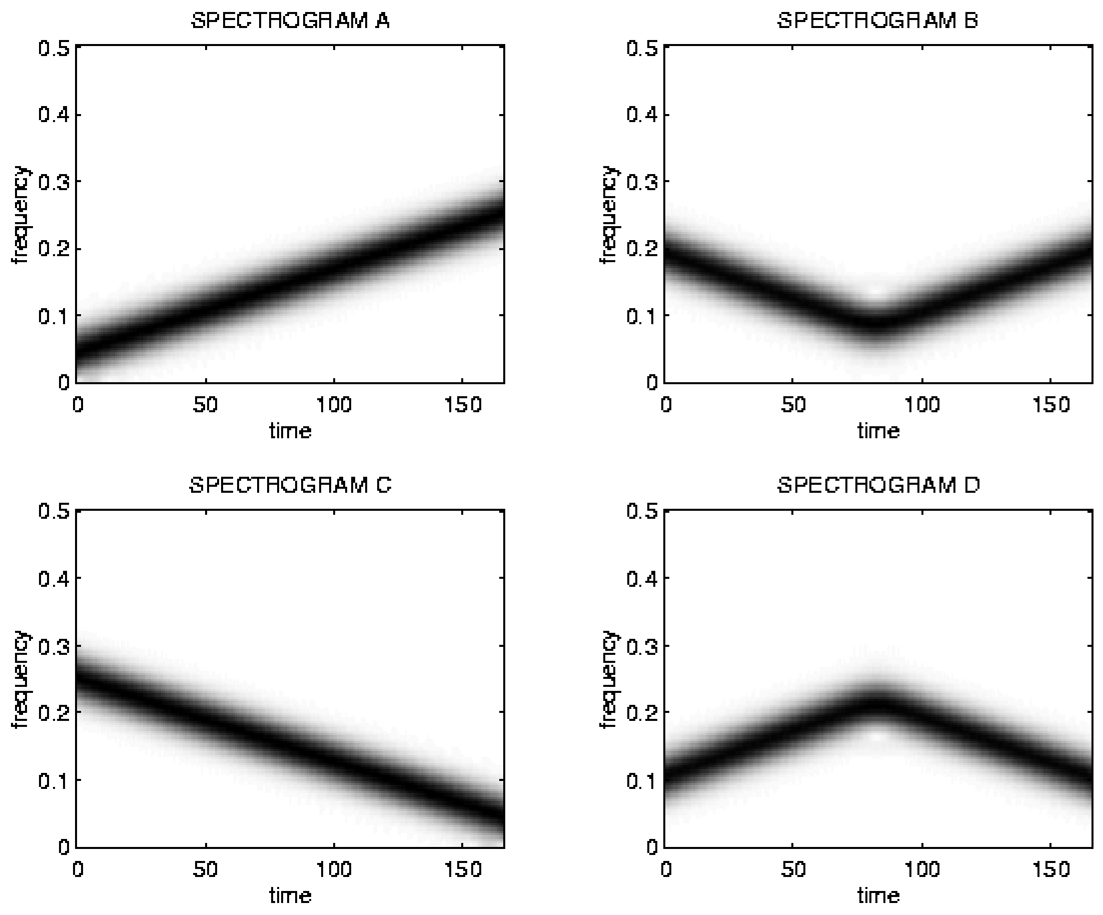
Problem: Match each signal to its spectrogram in [link].


SIGNAL 1
1
0.5
0
−0.5
−1
0
20
40
60
80
100
120
140
160
180
200


SIGNAL 2
1
0.5
0
−0.5
−1
0
20
40
60
80
100
120
140
160
180
200

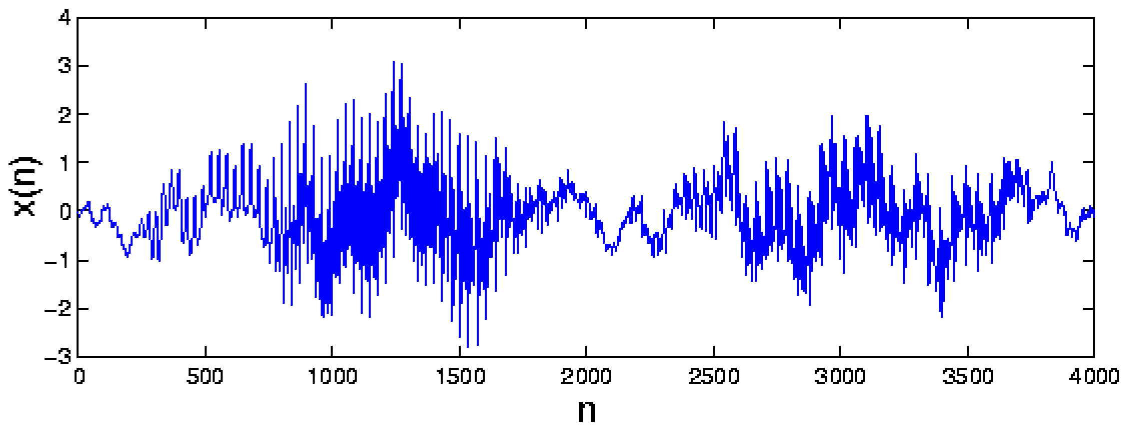

SIGNAL 3
1
0.5
0
−0.5
−1
0
20
40
60
80
100
120
140
160
180
200

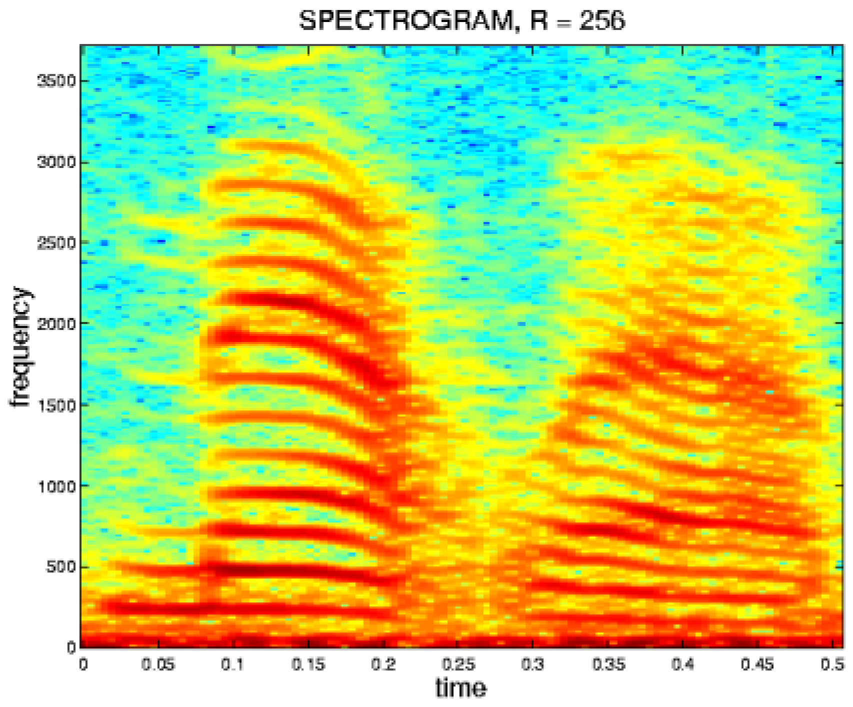

SIGNAL 4
1
0.5
0
−0.5
−1
0
20
40
60
80
100
120
140
160
180
200



Solution:

Spectrogram Example





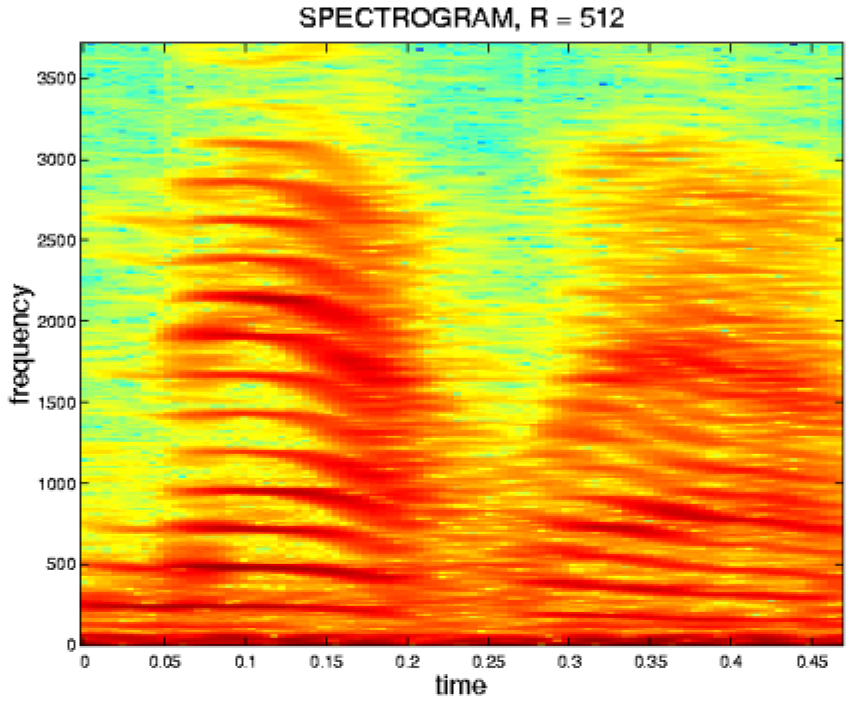
The matlab program for producing the figures above ([link] and [link]).

```
% LOAD DATA
load mtlb;
x = mtlb;
```

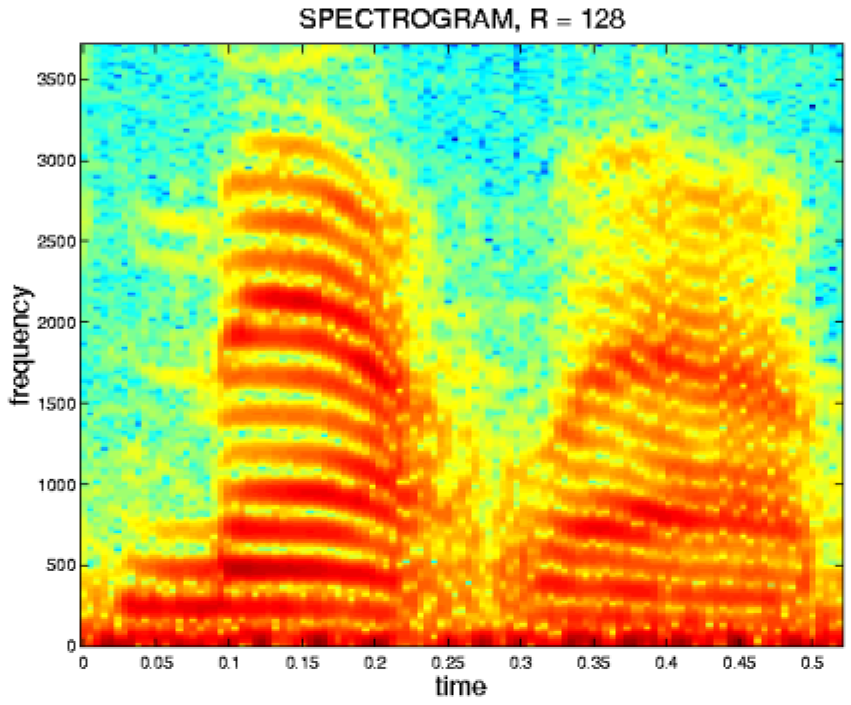
```
          figure(1), clf
          plot(0:4000,x)
          xlabel('n')
          ylabel('x(n)')

          % SET PARAMETERS
          R = 256;               % R: block length
          window = hamming(R);   % window function
of length R
          N = 512;               % N: frequency
discretization
          L = 35;                % L: time lapse
between blocks
          fs = 7418;             % fs: sampling
frequency
          overlap = R - L;

          % COMPUTE SPECTROGRAM
          [B,f,t] =
specgram(x,N,fs,window,overlap);

          % MAKE PLOT
          figure(2), clf
          imagesc(t,f,log10(abs(B)));
          colormap('jet')
          axis xy
          xlabel('time')
          ylabel('frequency')
          title('SPECTROGRAM, R = 256')
```

Effect of window length R

Narrow-band spectrogram: better frequency resolution

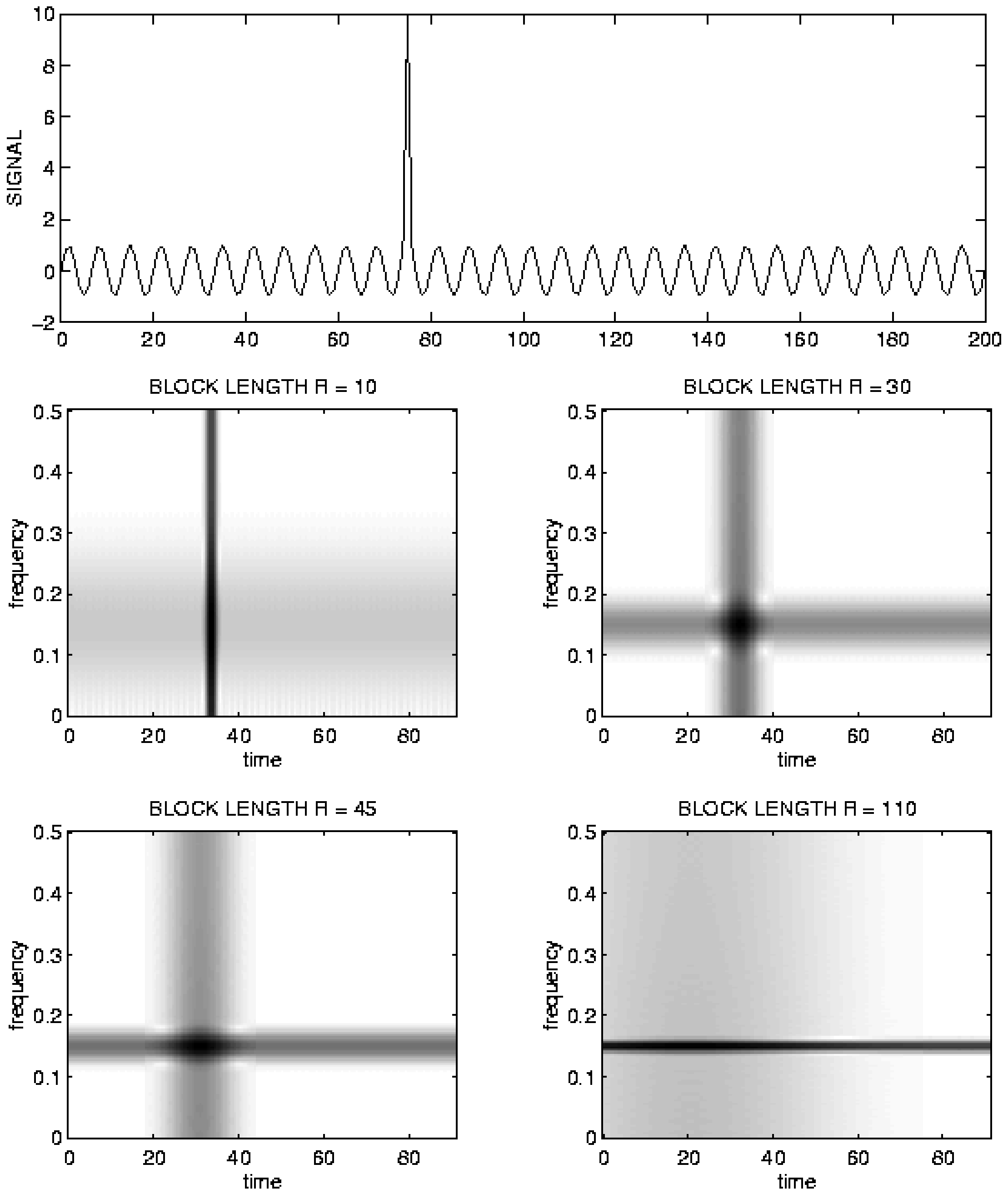


Wide-band spectrogram: better time resolution



Here is another example to illustrate the frequency/time resolution trade-off (See figures - [link], [link], and [link]).

Effect of Window Length R



Effect of L and N

A spectrogram is computed with different parameters:

$$L \in \{1, 10\}$$

$$N \in \{32, 256\}$$

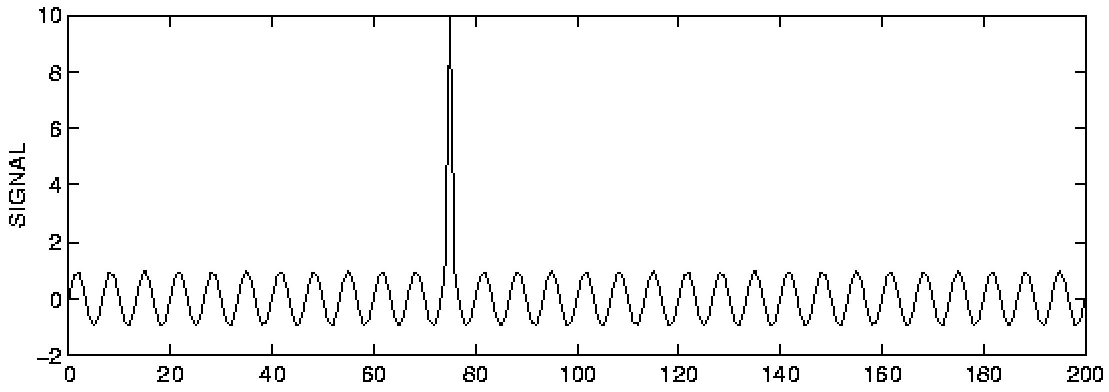
- L = time lapse between blocks.
- N = FFT length (Each block is zero-padded to length N.)

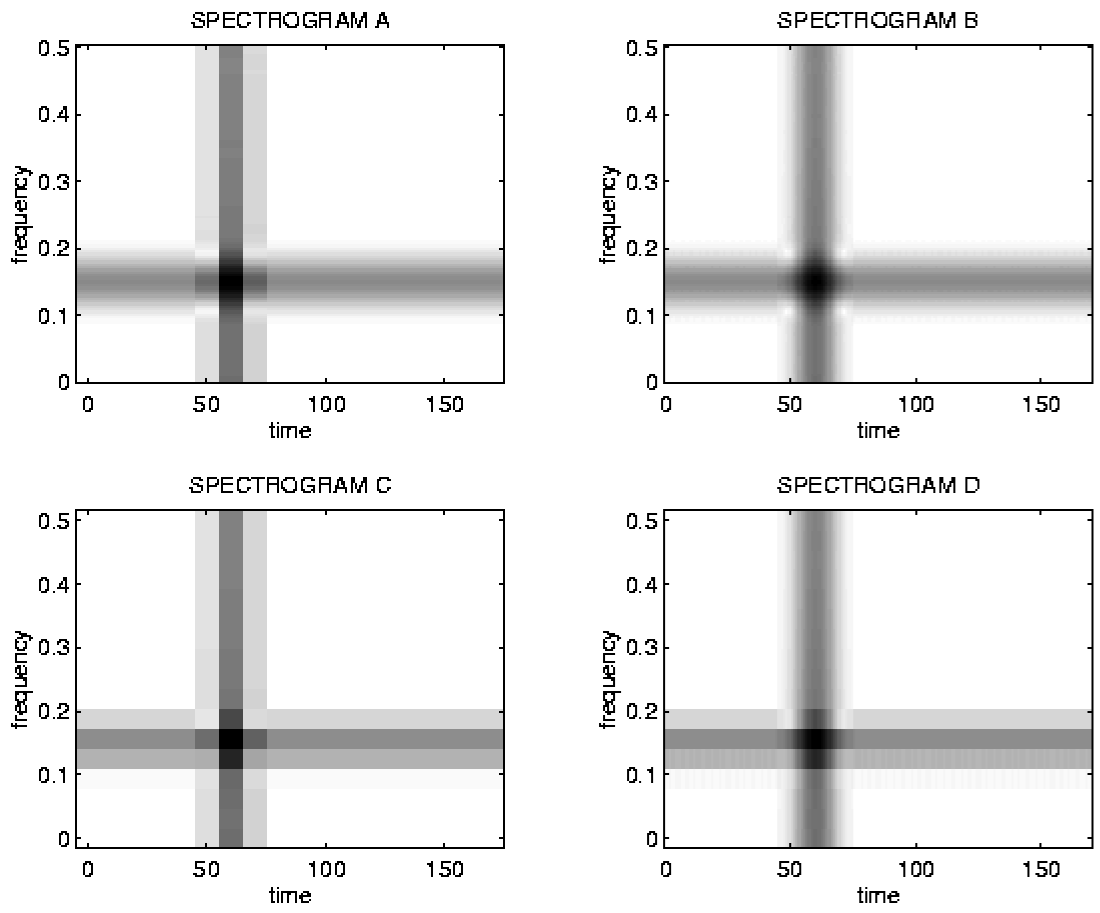
In each case, the block length is 30 samples.

Exercise:

Problem:

For each of the four spectrograms in [link] can you tell what L and N are?





Solution:

L and N do not effect the time resolution or the frequency resolution. They only affect the 'pixelation'.

Effect of R and L

Shown below are four spectrograms of the same signal. Each spectrogram is computed using a different set of parameters.

$$R \in \{120, 256, 1024\}$$

$$L \in \{35, 250\}$$

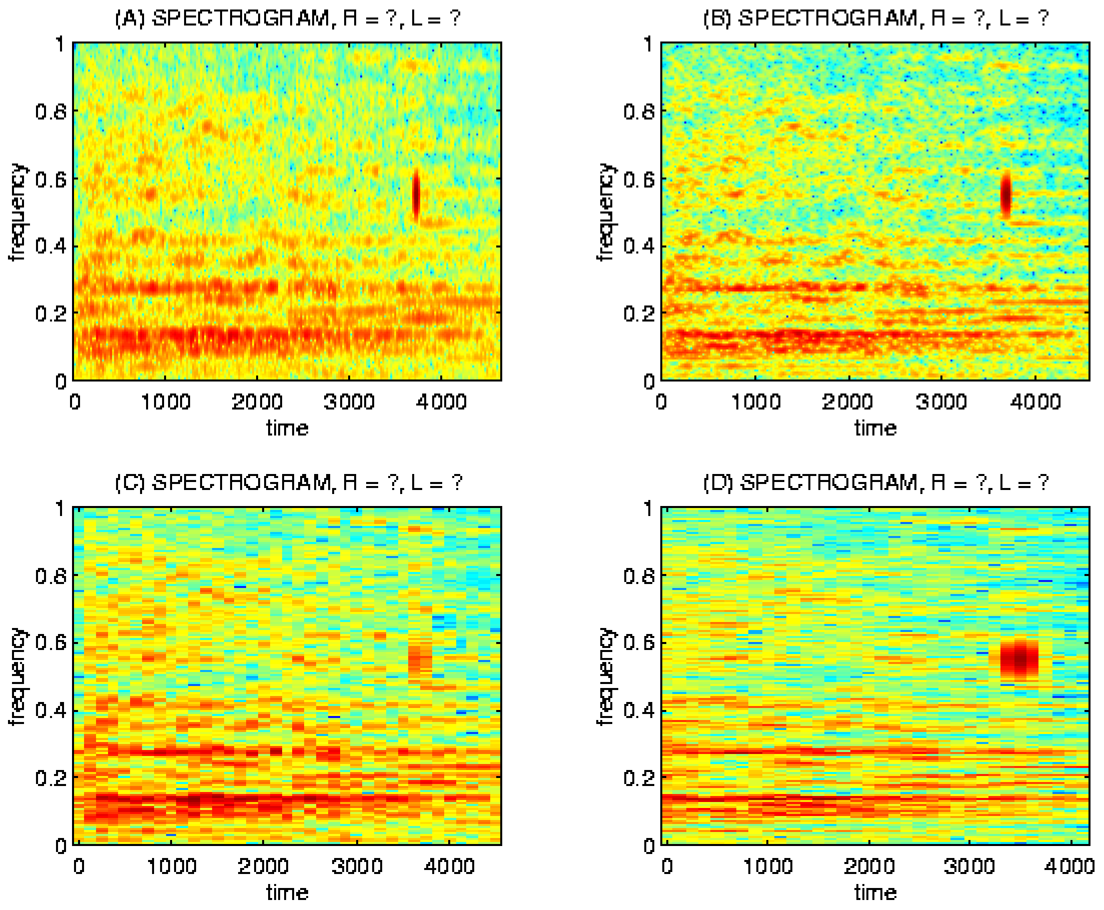
where

- R = block length
- L = time lapse between blocks.

Exercise:

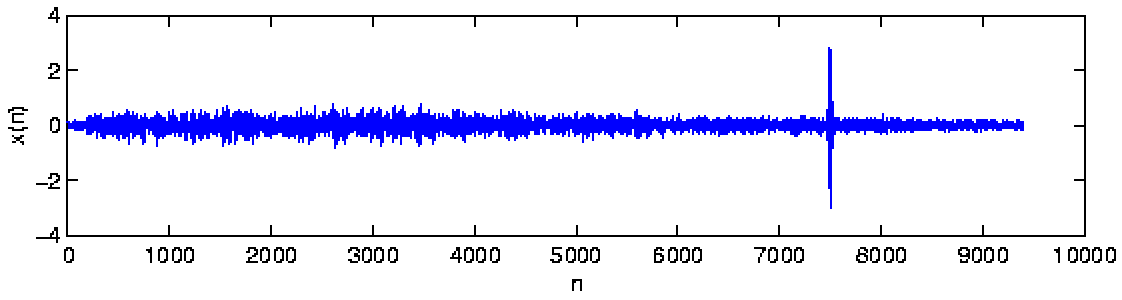
Problem:

For each of the four spectrograms in [link], match the above values of L and R.



Solution:

If you like, you may listen to this signal with the `soundsc` command; the data is in the file: `stft_data.m`. Here is a figure of the signal.



4
2
0
−2
−4
x(n)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
n

Overview of Fast Fourier Transform (FFT) Algorithms

A fast Fourier transform, or FFT, is not a new transform, but is a computationally efficient algorithm for the computing the DFT. The length-N DFT, defined as
Equation:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-\left(i \frac{2\pi nk}{N}\right)}$$

where $X(k)$ and $x(n)$ are in general complex-valued and $0 \leq k$, $n \leq N-1$, requires N complex multiplies to compute each $X(k)$. Direct computation of all N frequency samples thus requires N^2 complex multiplies and $N(N-1)$ complex additions. (This assumes precomputation of the DFT coefficients $W_N^{nk} \doteq e^{-\left(i \frac{2\pi nk}{N}\right)}$; otherwise, the cost is even higher.) For the large DFT lengths used in many applications, N^2 operations may be prohibitive. (For example, digital terrestrial television broadcast in Europe uses N = 2048 or 8192 OFDM channels, and the SETI project uses up to length-4194304 DFTs.) DFTs are thus almost always computed in practice by an FFT algorithm. FFTs are very widely used in signal processing, for applications such as spectrum analysis and digital filtering via fast convolution.

History of the FFT

It is now known that C.F. Gauss invented an FFT in 1805 or so to assist the computation of planetary orbits via discrete Fourier series. Various FFT algorithms were independently invented over the next two centuries, but FFTs achieved widespread awareness and impact only with the Cooley and Tukey algorithm published in 1965, which came at a time of increasing use of digital computers and when the vast range of applications of numerical Fourier techniques was becoming apparent. Cooley and Tukey's algorithm spawned a surge of research in FFTs and was also partly responsible for the emergence of Digital Signal Processing (DSP) as a distinct, recognized discipline. Since then, many different algorithms have been rediscovered or developed, and efficient FFTs now exist for all DFT lengths.

Summary of FFT algorithms

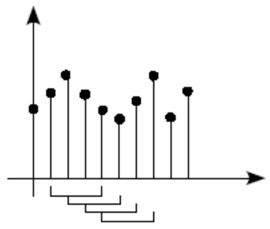
The main strategy behind most FFT algorithms is to factor a length-N DFT into a number of shorter-length DFTs, the outputs of which are reused multiple times (usually in additional short-length DFTs!) to compute the final results. The lengths of the short DFTs correspond to integer factors of the DFT length, N, leading to different algorithms for different lengths and factors. By far the most commonly used FFTs select $N = 2^M$ to be a power of two, leading to the very efficient power-of-two FFT algorithms, including the decimation-in-time radix-2 FFT and the decimation-in-frequency radix-2 FFT algorithms, the radix-4 FFT ($N = 4^M$), and the split-radix FFT. Power-of-two algorithms gain their high efficiency from extensive reuse of intermediate results and from the low complexity of length-2 and length-4 DFTs, which require no multiplications. Algorithms for lengths with repeated common factors (such as 2 or 4 in the radix-2 and radix-4 algorithms, respectively) require extra **twiddle factor** multiplications between the short-length DFTs, which together lead to a computational complexity of $O(N \log N)$, a very considerable savings over direct computation of the DFT.

The other major class of algorithms is the Prime-Factor Algorithms (PFA). In PFAs, the short-length DFTs must be of relatively prime lengths. These algorithms gain efficiency by reuse of intermediate computations and by eliminating twiddle-factor multiplies, but require more operations than the power-of-two algorithms to compute the short DFTs of various prime lengths. In the end, the computational costs of the prime-factor and the power-of-two algorithms are comparable for similar lengths, as illustrated in Choosing the Best FFT Algorithm. Prime-length DFTs cannot be factored into shorter DFTs, but in different ways both Rader's conversion and the chirp z-transform convert prime-length DFTs into convolutions of other lengths that can be computed efficiently using FFTs via fast convolution.

Some applications require only a few DFT frequency samples, in which case Goertzel's algorithm halves the number of computations relative to the DFT sum. Other applications involve successive DFTs of overlapped blocks of samples, for which the running FFT can be more efficient than separate FFTs of each block.

Running FFT

Some applications need DFT frequencies of the most recent N samples on an ongoing basis. One example is DTMF, or touch-tone telephone dialing, in which a detection circuit must constantly monitor the line for two simultaneous frequencies indicating that a telephone button is depressed. In such cases, most of the data in each successive block of samples is the same, and it is possible to efficiently update the DFT value from the previous sample to compute that of the current sample. [link] illustrates successive length-4 blocks of data for which successive DFT values may be needed. The **running FFT** algorithm described here can be used to compute successive DFT values at a cost of only two complex multiplies and additions per DFT frequency.



The running FFT efficiently computes DFT values for successive overlapped blocks of samples.

The running FFT algorithm is derived by expressing each DFT sample, $X_{n+1}(\omega_k)$, for the next block at time $n+1$ in terms of the previous value, $X_n(\omega_k)$, at time n.

$$X_n(\omega_k) = \sum_{p=0}^{N-1} x(n-p)e^{-(i\omega_k p)}$$

$$X_{n+1}(\omega_k) = \sum_{p=0}^{N-1} x(n+1-p)e^{-(i\omega_k p)}$$

Let $q = p - 1$:

$$X_{n+1}(\omega_k) = \sum_{q=-1}^{N-2} x(n-q)e^{-(i\omega_k(q-1))} = e^{i\omega_k} \sum_{q=0}^{N-2} x(n-q)e^{-(i\omega_k q)} + x(n+1)$$

Now let's add and subtract $e^{-(i\omega_k(N-2))}x(n-N+1)$:

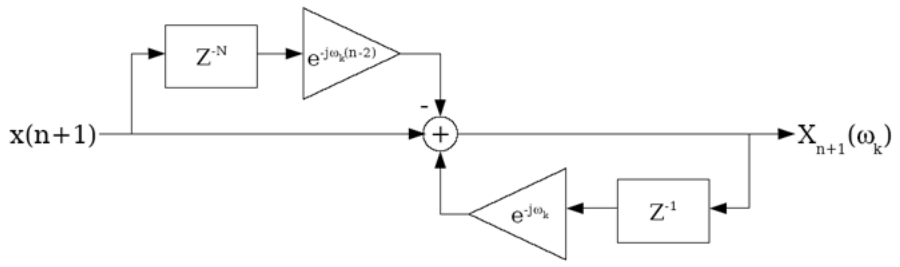
Equation:

$$\begin{aligned} X_{n+1}(\omega_k) &= e^{i\omega_k} \sum_{q=0}^{N-2} x(n-q)e^{-(i\omega_k q)} + e^{i\omega_k} x(n-(N-1))e^{-(i\omega_k(N-1))} - e^{-(i\omega_k(N-2))}x(n-N+1) + \\ &= e^{i\omega_k} \sum_{q=0}^{N-1} x(n-q)e^{-(i\omega_k)} + x(n+1) - e^{-(i\omega_k)}x(n-N+1) \\ &= e^{i\omega_k} X_n(\omega_k) + x(n+1) - e^{-(i\omega_k(N-2))}x(n-N+1) \end{aligned}$$

This running FFT algorithm requires only two complex multiplies and adds per update, rather than N if each DFT value were recomputed according to the DFT equation. Another advantage of this algorithm is that it works for

any ω_k, rather than just the standard DFT frequencies. This can make it advantageous for applications, such as DTMF detection, where only a few arbitrary frequencies are needed.

Successive computation of a specific DFT frequency for overlapped blocks can also be thought of as a length-N FIR filter. The running FFT is an efficient recursive implementation of this filter for this special case. [link] shows a block diagram of the running FFT algorithm. The running FFT is one way to compute DFT filterbanks. If a window other than rectangular is desired, a running FFT requires either a fast recursive implementation of the corresponding windowed, modulated impulse response, or it must have few non-zero coefficients so that it can be applied after the running FFT update via frequency-domain convolution. DFT-symmmetric raised-cosine windows are an example.



Block diagram of the running FFT computation, implemented as a recursive filter

Goertzel's Algorithm

Some applications require only a few DFT frequencies. One example is frequency-shift keying (FSK) demodulation, in which typically two frequencies are used to transmit binary data; another example is DTMF, or touch-tone telephone dialing, in which a detection circuit must constantly monitor the line for two simultaneous frequencies indicating that a telephone button is depressed. Goertzel's algorithm reduces the number of real-valued multiplications by almost a factor of two relative to direct computation via the DFT equation. Goertzel's algorithm is thus useful for computing a **few** frequency values; if many or most DFT values are needed, FFT algorithms that compute all DFT samples in $O(N\log N)$ operations are faster. Goertzel's algorithm can be derived by converting the DFT equation into an equivalent form as a convolution, which can be efficiently implemented as a digital filter. For increased clarity, in the equations below the complex exponential is denoted as $e^{-\left(i\frac{2\pi k}{N}\right)}=W_N^k$. Note that because W_N^{-Nk} always equals 1, the DFT equation can be rewritten as a convolution, or filtering operation:

Equation:

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n)1W_N^{nk} \\ &= \sum_{n=0}^{N-1} x(n)W_N^{-Nk}W_N^{nk} \\ &= \sum_{n=0}^{N-1} x(n)W_N^{(N-n)(-k)} \\ &= \left(\left(\left(W_N^{-k}x(0)+x(1)\right)W_N^{-k}+x(2)\right)W_N^{-k}+\ldots+x(N-1)\right)W_N^{-k} \end{aligned}$$

Note that this last expression can be written in terms of a recursive difference equation

$$y(n)=W_N^{-k}y(n-1)+x(n)$$

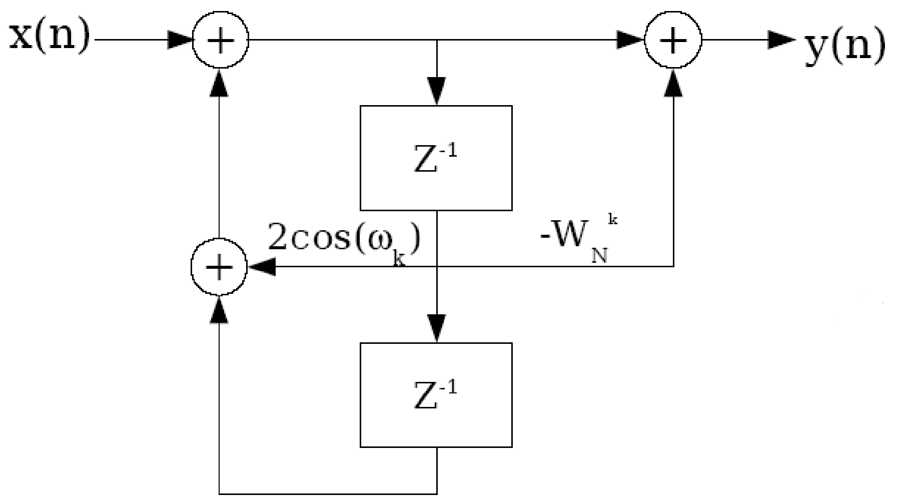
where $y(-1)=0$. The DFT coefficient equals the output of the difference equation at time $n=N$:

$$X(k)=y(N)$$

Expressing the difference equation as a z-transform and multiplying both numerator and denominator by $1-W_N^kz^{-1}$ gives the transfer function

$$\frac{Y(z)}{X(z)}=H(z)=\frac{1}{1-W_N^{-k}z^{-1}}=\frac{1-W_N^kz^{-1}}{1-\left(\left(W_N^k+W_N^{-k}\right)z^{-1}-z^{-2}\right)}=\frac{1-W_N^kz^{-1}}{1-\left(2\cos\left(\frac{2\pi k}{N}\right)z^{-1}-z^{-2}\right)}$$

This system can be realized by the structure in [link]



We want $y(n)$ not for all n, but only for $n = N$. We can thus compute only the **recursive** part, or just the left side of the flow graph in [link], for $n = [0, 1, \ldots, N]$, which involves only a **real/complex** product rather than a complex/complex product as in a direct DFT, plus one complex multiply to get $y(N) = X(k)$.

Note: The input $x(N)$ at time $n = N$ must equal 0! A slightly more efficient alternate implementation that computes the full recursion only through $n = N - 1$ and combines the nonzero operations of the final recursion with the final complex multiply can be found here, complete with pseudocode (for real-valued data).

If the data are real-valued, only real/real multiplications and real additions are needed until the final multiply.

Note: The computational cost of Goertzel's algorithm is thus $2N + 2$ real multiplies and $4N - 2$ real adds, a reduction of almost a factor of two in the number of real multiplies relative to direct computation via the DFT equation. If the data are real-valued, this cost is almost halved again.

For certain frequencies, additional simplifications requiring even fewer multiplications are possible. (For example, for the DC ($k = 0$) frequency, all the multipliers equal 1 and only additions are needed.) A correspondence by C.G. Boncelet, Jr. describes some of these additional simplifications. Once again, Goertzel's and Boncelet's algorithms are efficient for a few DFT frequency samples; if more than $\log N$ frequencies are needed, $O(N \log N)$ FFT algorithms that compute all frequencies simultaneously will be more efficient.

Power-of-two FFTs

FFTs of length $N = 2^M$ equal to a power of two are, by far, the most commonly used. These algorithms are very efficient, relatively simple, and a single program can compute power-of-two FFTs of different lengths. As with most FFT algorithms, they gain their efficiency by computing **all** DFT points simultaneously through extensive reuse of intermediate computations; they are thus efficient when many DFT frequency samples are needed. The simplest power-of-two FFTs are the decimation-in-time radix-2 FFT and the decimation-in-frequency radix-2 FFT; they reduce the length-$N = 2^M$ DFT to a series of length-2 DFT computations with **twiddle-factor** complex multiplications between them. The radix-4 FFT algorithm similarly reduces a length-$N = 4^M$ DFT to a series of length-4 DFT computations with twiddle-factor multiplies in between. Radix-4 FFTs require only 75% as many complex multiplications as the radix-2 algorithms, although the number of complex additions remains the same. Radix-8 and higher-radix FFT algorithms can be derived using multi-dimensional index maps to reduce the computational complexity a bit more. However, the split-radix algorithm and its recent extensions combine the best elements of the radix-2 and radix-4 algorithms to obtain lower complexity than either or than any higher radix, requiring only two-thirds as many complex multiplies as the radix-2 algorithms. All of these algorithms obtain huge savings over direct computation of the DFT, reducing the complexity from $O\left(N^2\right)$ to $O(N \log N)$.

The efficiency of an FFT implementation depends on more than just the number of computations. Efficient FFT programming tricks can make up to a several-fold difference in the run-time of FFT programs. Alternate FFT structures can lead to a more convenient data flow for certain hardware. As discussed in choosing the best FFT algorithm, certain hardware is designed for, and thus most efficient for, FFTs of specific lengths or radices.

Decimation-in-time (DIT) Radix-2 FFT

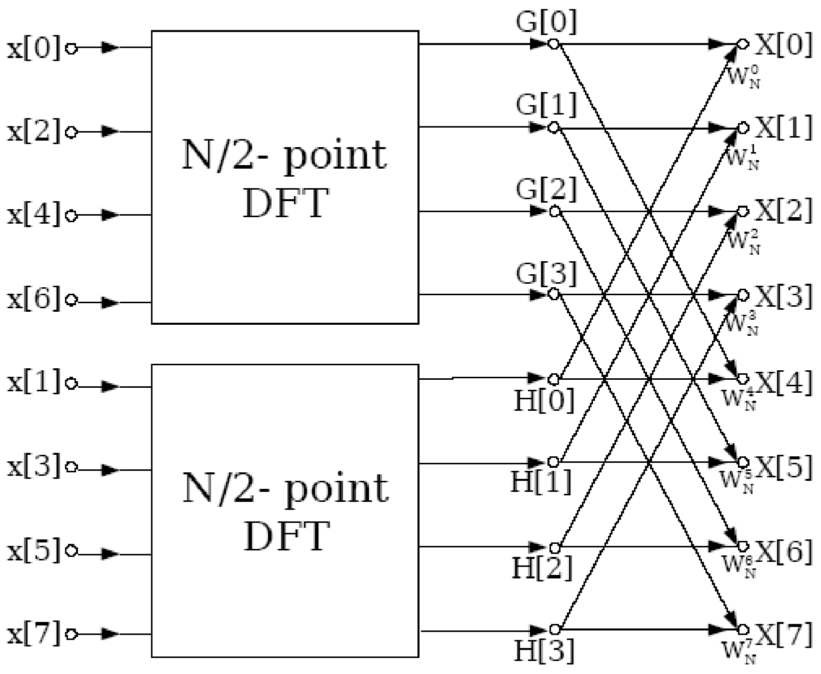
The radix-2 decimation-in-time and decimation-in-frequency fast Fourier transforms (FFTs) are the simplest FFT algorithms. Like all FFTs, they gain their speed by reusing the results of smaller, intermediate computations to compute multiple DFT frequency outputs.

Decimation in time

The radix-2 decimation-in-time algorithm rearranges the discrete Fourier transform (DFT) equation into two parts: a sum over the even-numbered discrete-time indices $n=[0,2,4,\ldots,N-2]$ and a sum over the odd-numbered indices $n=[1,3,5,\ldots,N-1]$ as in [link]:
Equation:

$$\begin{aligned}
X(k) &= \sum_{n=0}^{N-1} x(n)e^{-\left(i\frac{2\pi nk}{N}\right)} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-\left(i\frac{2\pi\times(2n)k}{N}\right)} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)e^{-\left(i\frac{2\pi(2n+1)k}{N}\right)} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-\left(i\frac{2\pi nk}{\frac{N}{2}}\right)} + e^{-\left(i\frac{2\pi k}{N}\right)} \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)e^{-\left(i\frac{2\pi nk}{\frac{N}{2}}\right)} \\
&= \mathrm{DFT}_{\frac{N}{2}}\left[[x(0),x(2),\ldots,x(N-2)]\right] + W_N^k\,\mathrm{DFT}_{\frac{N}{2}}\left[[x(1),x(3),\ldots,x(N-1)]\right]
\end{aligned}$$

The mathematical simplifications in [link] reveal that all DFT frequency outputs $X(k)$ can be computed as the sum of the outputs of two length-$\frac{N}{2}$ DFTs, of the even-indexed and odd-indexed discrete-time samples, respectively, where the odd-indexed short DFT is multiplied by a so-called **twiddle factor** term $W_N^k = e^{-\left(i\frac{2\pi k}{N}\right)}$. This is called a **decimation in time** because the time samples are rearranged in alternating groups, and a **radix-2** algorithm because there are two groups. [link] graphically illustrates this form of the DFT computation, where for convenience the frequency outputs of the length-$\frac{N}{2}$ DFT of the even-indexed time samples are denoted $G(k)$ and those of the odd-indexed samples as $H(k)$. Because of the periodicity with $\frac{N}{2}$ frequency samples of these length-$\frac{N}{2}$ DFTs, $G(k)$ and $H(k)$ can be used to compute **two** of the length-N DFT frequencies, namely $X(k)$ and $X\left(k+\frac{N}{2}\right)$, but with a different twiddle factor. This reuse of these short-length DFT outputs gives the FFT its computational savings.



Decimation in time of a length-N DFT into two length-$\frac{N}{2}$ DFTs followed by a combining stage.

Whereas direct computation of all N DFT frequencies according to the DFT equation would require N^2 complex multiplies and $N^2 - N$ complex additions (for complex-valued data), by reusing the results of the two short-length DFTs as illustrated in [link], the computational cost is now

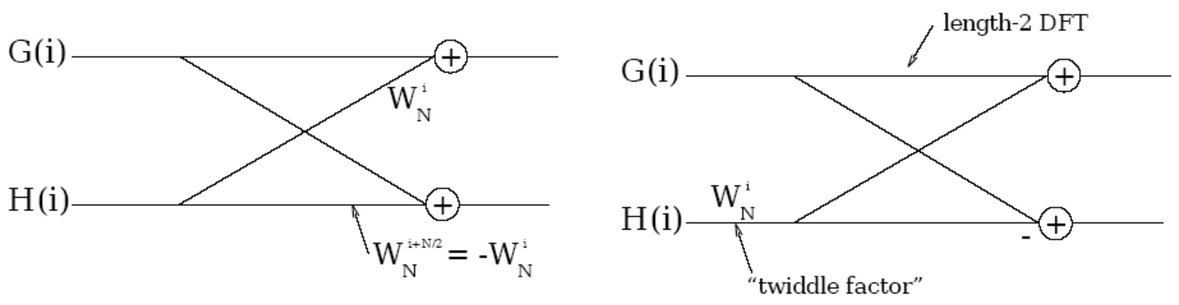
New Operation Counts

- $2\left(\frac{N}{2}\right)^2 + N = \frac{N^2}{2} + N$ complex multiplies
- $2\frac{N}{2}\left(\frac{N}{2} - 1\right) + N = \frac{N^2}{2}$ complex additions

This simple reorganization and reuse has reduced the total computation by almost a factor of two over direct DFT computation!

Additional Simplification

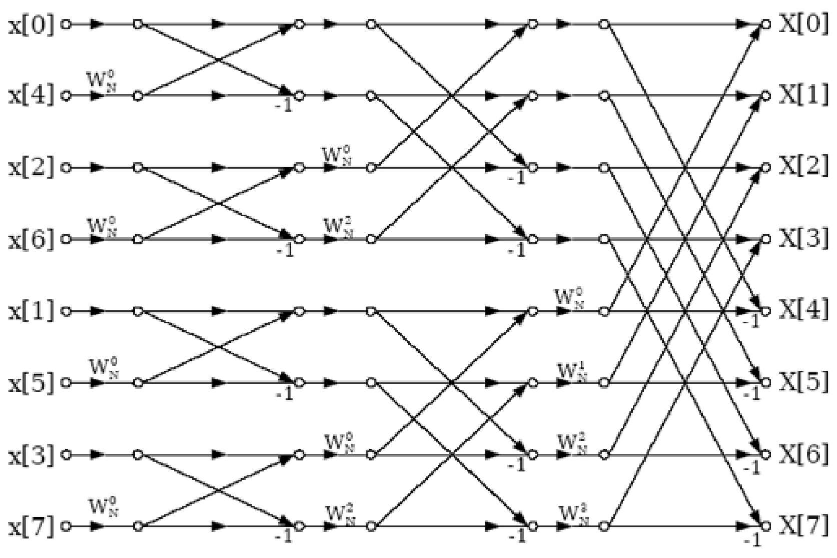
A basic **butterfly** operation is shown in [link], which requires only $\frac{N}{2}$ **twiddle-factor** multiplies per **stage**. It is worthwhile to note that, after merging the twiddle factors to a single term on the lower branch, the remaining butterfly is actually a length-2 DFT! The theory of multi-dimensional index maps shows that this must be the case, and that FFTs of any factorable length may consist of successive stages of shorter-length FFTs with twiddle-factor multiplications in between.



Radix-2 DIT butterfly simplification: both operations produce the same outputs

Radix-2 decimation-in-time FFT

The same radix-2 decimation in time can be applied recursively to the two length $\frac{N}{2}$ DFTs to save computation. When successively applied until the shorter and shorter DFTs reach length-2, the result is the radix-2 DIT FFT algorithm.



Radix-2 Decimation-in-Time FFT algorithm for a length-8 signal

The full radix-2 decimation-in-time decomposition illustrated in [link] using the simplified butterflies involves $M=\log_2 N$ stages, each with $\frac{N}{2}$ butterflies per stage. Each butterfly requires 1 complex multiply and 2 adds per butterfly. The total cost of the algorithm is thus
Computational cost of radix-2 DIT FFT

- $\frac{N}{2}\log_2 N$ complex multiplies
- $N\log_2$ N complex adds

This is a remarkable savings over direct computation of the DFT. For example, a length-1024 DFT would require 1048576 complex multiplications and 1047552 complex additions with direct computation, but only 5120 complex multiplications and 10240 complex additions using the radix-2 FFT, a savings by a factor of 100 or more. The relative savings increase with longer FFT lengths, and are less for shorter lengths.

Modest additional reductions in computation can be achieved by noting that certain twiddle factors, namely Using special butterflies for W_N^0, $W_N^{\frac{N}{2}}$, $W_N^{\frac{N}{4}}$, $W_N^{\frac{N}{8}}$, $W_N^{\frac{3N}{8}}$, require no multiplications, or fewer real multiplies than other ones. By implementing special butterflies for these twiddle factors as discussed in FFT algorithm and programming tricks, the computational cost of the radix-2 decimation-in-time FFT can be reduced to

- $2N\log_2 N-7N+12$ real multiplies
- $3N\log_2 N-3N+4$ real additions

Note: In a decimation-in-time radix-2 FFT as illustrated in [link], the input is in **bit-reversed** order (hence "decimation-in-time"). That is, if the time-sample index n is written as a binary number, the order is that binary number reversed. The bit-reversal process is illustrated for a length-$N=8$ example below.

Example:
N=8

In-order index	In-order index in binary	Bit-reversed binary	Bit-reversed index

In-order index	**In-order index in binary**	**Bit-reversed binary**	**Bit-reversed index**
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

It is important to note that, if the input signal data are placed in bit-reversed order before beginning the FFT computations, the outputs of each butterfly throughout the computation can be placed in the same memory locations from which the inputs were fetched, resulting in an **in-place algorithm** that requires no extra memory to perform the FFT. Most FFT implementations are in-place, and overwrite the input data with the intermediate values and finally the output.

Example FFT Code

The following function, written in the C programming language, implements a radix-2 decimation-in-time FFT. It is designed for computing the DFT of complex-valued inputs to produce complex-valued outputs, with the real and imaginary parts of each number stored in separate double-precision floating-point arrays. It is an in-place algorithm, so the intermediate and final output values are stored in the same array as the input data, which is overwritten. After initializations, the program first bit-reverses the discrete-time samples, as is typical with a decimation-in-time algorithm (but see alternate FFT structures for DIT algorithms with other input orders), then computes the FFT in stages according to the above description.

Ihis FFT program uses a standard three-loop structure for the main FFT computation. The outer loop steps through the stages (each column in [link]); the middle loop steps through "**flights**" (butterflies with the same twiddle factor from each short-length DFT at each stage), and the inner loop steps through the individual butterflies. This ordering minimizes the number of fetches or computations of the twiddle-factor values. Since the bit-reverse of a bit-reversed index is the original index, bit-reversal can be performed fairly simply by swapping pairs of data.

Note: While of $O(N \log N)$ complexity and thus much faster than a direct DFT, this simple program is optimized for clarity, not for speed. A speed-optimized program making use of additional efficient FFT algorithm and programming tricks will compute a DFT several times faster on most machines.

```
/**********************************************************/
/* fft.c                                                  */
/* (c) Douglas L. Jones                                   */
/* University of Illinois at Urbana-Champaign             */
/* January 19, 1992                                       */
/*                                                        */
/*   fft: in-place radix-2 DIT DFT of a complex input     */
/*                                                        */
/*   input:                                               */
/* n: length of FFT: must be a power of two               */
/* m: n = 2**m                                            */
/*   input/output                                         */
/* x: double array of length n with real part of data     */
/* y: double array of length n with imag part of data     */
/*                                                        */
/*   Permission to copy and use this program is granted   */
/*   under a Creative Commons "Attribution" license       */
/*   http://creativecommons.org/licenses/by/1.0/          */
/**********************************************************/
fft(n,m,x,y)
int n,m;
double x[],y[];
{
int i,j,k,n1,n2;
double c,s,e,a,t1,t2;


j = 0; /* bit-reverse */
n2 = n/2;
for (i=1; i < n - 1; i++)
{
  n1 = n2;
  while ( j >= n1 )
```

```
      {
       j = j - n1;
       n1 = n1/2;
      }
    j = j + n1;

    if (i < j)
      {
       t1 = x[i];
       x[i] = x[j];
       x[j] = t1;
       t1 = y[i];
       y[i] = y[j];
       y[j] = t1;
      }
  }


  n1 = 0; /* FFT */
  n2 = 1;

  for (i=0; i < m; i++)
  {
    n1 = n2;
    n2 = n2 + n2;
    e = -6.283185307179586/n2;
    a = 0.0;

    for (j=0; j < n1; j++)
      {
       c = cos(a);
       s = sin(a);
       a = a + e;

       for (k=j; k < n; k=k+n2)
         {
          t1 = c*x[k+n1] - s*y[k+n1];
          t2 = s*x[k+n1] + c*y[k+n1];
          x[k+n1] = x[k] - t1;
          y[k+n1] = y[k] - t2;
          x[k] = x[k] + t1;
          y[k] = y[k] + t2;
         }
      }
  }
```

```
return;
}
```

Decimation-in-Frequency (DIF) Radix-2 FFT

The radix-2 decimation-in-frequency and decimation-in-time fast Fourier transforms (FFTs) are the simplest FFT algorithms. Like all FFTs, they compute the discrete Fourier transform (DFT)

Equation:

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) e^{-\left(i \frac{2\pi nk}{N}\right)} \\ &= \sum_{n=0}^{N-1} x(n) W_N^{nk} \end{aligned}$$

where for notational convenience $W_N^k = e^{-\left(i\frac{2\pi k}{N}\right)}$. FFT algorithms gain their speed by reusing the results of smaller, intermediate computations to compute multiple DFT frequency outputs.

Decimation in frequency

The radix-2 decimation-in-frequency algorithm rearranges the discrete Fourier transform (DFT) equation into two parts: computation of the even-numbered discrete-frequency indices $X(k)$ for $k = [0, 2, 4, \ldots, N-2]$ (or $X(2r)$ as in [link]) and computation of the odd-numbered indices $k = [1, 3, 5, \ldots, N-1]$ (or $X(2r+1)$ as in [link])

Equation:

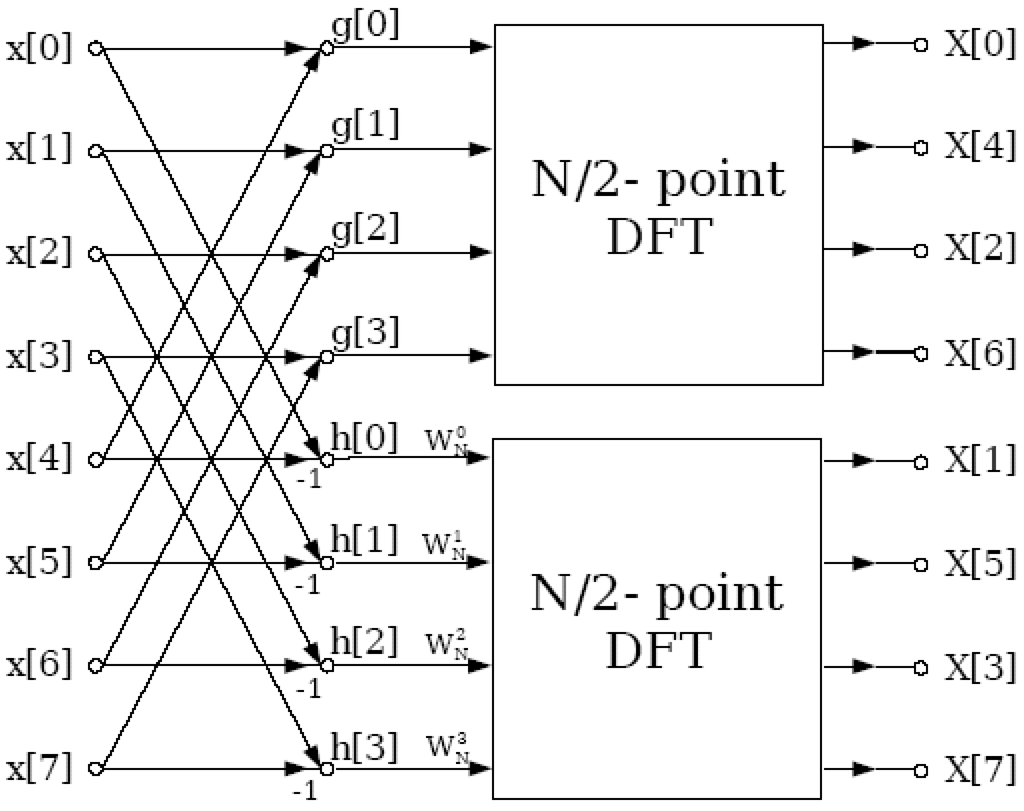
$$\begin{aligned} X(2r) &= \sum_{n=0}^{N-1} x(n) W_N^{2rn} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(n) W_N^{2rn} + \sum_{n=0}^{\frac{N}{2}-1} x\left(n + \frac{N}{2}\right) W_N^{2r\left(n+\frac{N}{2}\right)} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(n) W_N^{2rn} + \sum_{n=0}^{\frac{N}{2}-1} x\left(n + \frac{N}{2}\right) W_N^{2rn} 1 \\ &= \sum_{n=0}^{\frac{N}{2}-1} \left(x(n) + x\left(n + \frac{N}{2}\right)\right) W_{\frac{N}{2}}^{rn} \\ &= \text{DFT}_{\frac{N}{2}} \left[x(n) + x\left(n + \frac{N}{2}\right)\right] \end{aligned}$$

Equation:

$$\begin{aligned} X(2r+1) &= \sum_{n=0}^{N-1} x(n) W_N^{(2r+1)n} \\ &= \sum_{n=0}^{\frac{N}{2}-1} \left(x(n) + W_N^{\frac{N}{2}} x\left(n+\frac{N}{2}\right)\right) W_N^{(2r+1)n} \\ &= \sum_{n=0}^{\frac{N}{2}-1} \left(\left(x(n) - x\left(n+\frac{N}{2}\right)\right) W_N^n\right) W_{\frac{N}{2}}^{rn} \\ &= \mathbf{DFT}_{\frac{N}{2}}\left[\left(x(n) - x\left(n+\frac{N}{2}\right)\right) W_N^n\right] \end{aligned}$$

The mathematical simplifications in [link] and [link] reveal that both the even-indexed and odd-indexed frequency outputs $X(k)$ can each be computed by a length-$\frac{N}{2}$ DFT. The inputs to these DFTs are sums or differences of the first and second halves of the input signal, respectively, where the input to the short DFT producing the odd-indexed frequencies is multiplied by a so-called **twiddle factor** term $W_N^k = e^{-\left(i\frac{2\pi k}{N}\right)}$. This is called a **decimation in frequency** because the frequency samples are computed separately in alternating groups, and a **radix-2** algorithm because there are two groups. [link] graphically illustrates this form of the DFT computation. This conversion of the full DFT into a series of shorter DFTs with a simple preprocessing step gives the decimation-in-frequency FFT its computational savings.

Decimation in frequency of a length-N DFT into two length-$\frac{N}{2}$ DFTs preceded by a preprocessing stage.



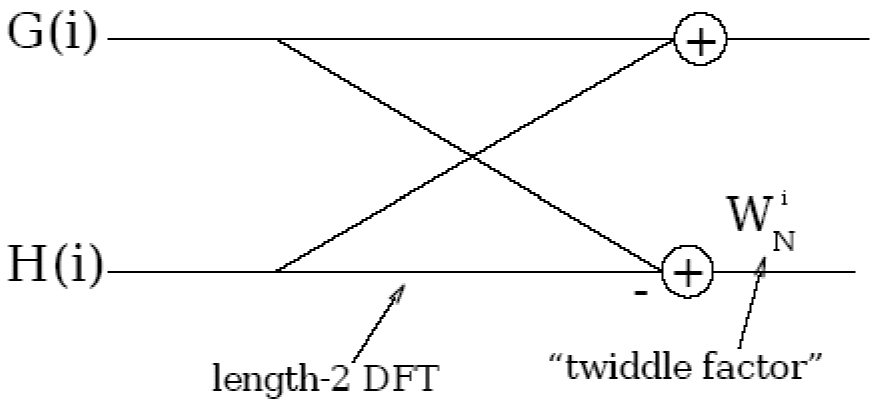
Whereas direct computation of all N DFT frequencies according to the DFT equation would require N^2 complex multiplies and $N^2 - N$ complex additions (for complex-valued data), by breaking the computation into two short-length DFTs with some preliminary combining of the data, as illustrated in [link], the computational cost is now

New Operation Counts

- $2\left(\frac{N}{2}\right)^2 + N = \frac{N^2}{2} + \frac{N}{2}$ complex multiplies
- $2\frac{N}{2}\left(\frac{N}{2} - 1\right) + N = \frac{N^2}{2}$ complex additions

This simple manipulation has reduced the total computational cost of the DFT by almost a factor of two!

The initial combining operations for both short-length DFTs involve parallel groups of two time samples, $x(n)$ and $x\left(n + \frac{N}{2}\right)$. One of these so-called **butterfly** operations is illustrated in [link]. There are $\frac{N}{2}$ butterflies per **stage**, each requiring a complex addition and subtraction followed by one **twiddle-factor** multiplication by $W_N^n = e^{-\left(i\frac{2\pi n}{N}\right)}$ on the lower output branch.

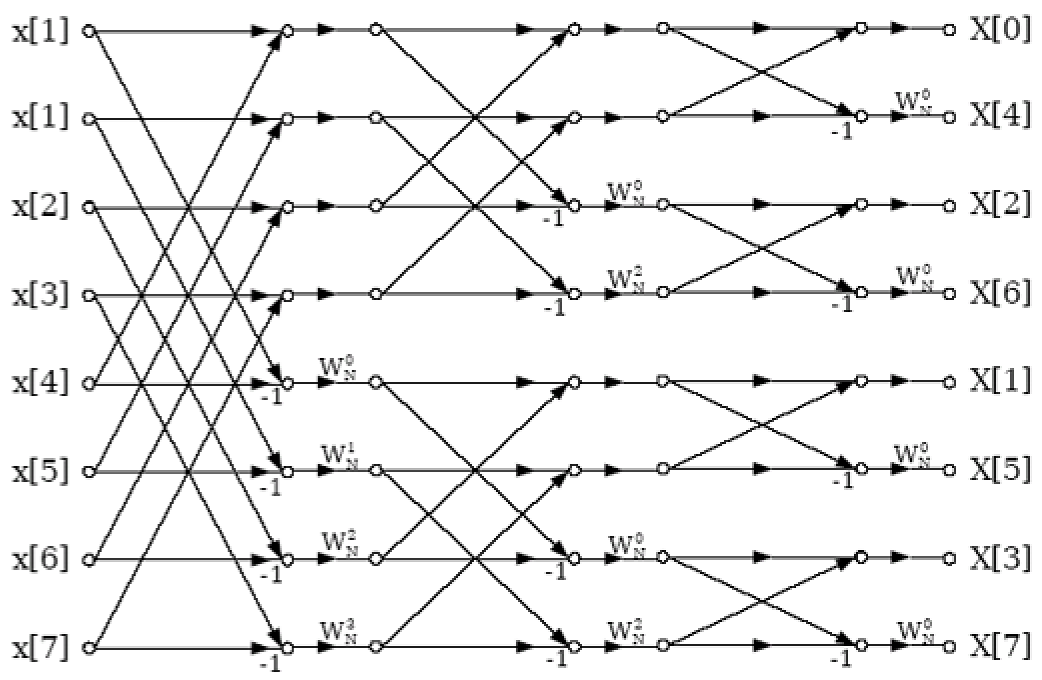


DIF butterfly: twiddle factor after length-2 DFT

It is worthwhile to note that the initial add/subtract part of the DIF butterfly is actually a length-2 DFT! The theory of multi-dimensional index maps shows that this must be the case, and that FFTs of any factorable length may consist of successive stages of shorter-length FFTs with twiddle-factor multiplications in between. It is also worth noting that this butterfly differs from the decimation-in-time radix-2 butterfly in that the twiddle factor multiplication occurs **after** the combining.

Radix-2 decimation-in-frequency algorithm

The same radix-2 decimation in frequency can be applied recursively to the two length-$\frac{N}{2}$ DFTs to save additional computation. When successively applied until the shorter and shorter DFTs reach length-2, the result is the radix-2 decimation-in-frequency FFT algorithm.



Radix-2 decimation-in-frequency FFT for a length-8 signal

The full radix-2 decimation-in-frequency decomposition illustrated in [link] requires $M=\log_2 N$ stages, each with $\frac{N}{2}$ butterflies per stage. Each butterfly requires 1 complex multiply and 2 adds per butterfly. The total cost of the algorithm is thus

Computational cost of radix-2 DIF FFT

- $\frac{N}{2}\log_2 N$ complex multiplies
- $N\log_2$ N complex adds

This is a remarkable savings over direct computation of the DFT. For example, a length-1024 DFT would require 1048576 complex multiplications and 1047552 complex additions with direct computation, but only 5120 complex multiplications and 10240 complex additions using the radix-2 FFT, a savings by a factor of 100 or more. The relative savings increase with longer FFT lengths, and are less for shorter lengths. Modest additional reductions in computation can be achieved by noting that certain twiddle factors, namely $W_N^0, W_N^{\frac{N}{2}}, W_N^{\frac{N}{4}}, W_N^{\frac{N}{8}}, W_N^{\frac{3N}{8}}$, require no multiplications, or fewer real multiplies than other ones. By implementing special butterflies for these twiddle factors as discussed in FFT algorithm and programming tricks, the computational cost of the radix-2 decimation-in-frequency FFT can be reduced to

- $2N\log_2 N - 7N + 12$ real multiplies
- $3N\log_2 N - 3N + 4$ real additions

The decimation-in-frequency FFT is a flow-graph reversal of the decimation-in-time FFT: it has the same twiddle factors (in reverse pattern) and the same operation counts.

Note: In a decimation-in-frequency radix-2 FFT as illustrated in [link], the output is in **bit-reversed** order (hence "decimation-in-frequency"). That is, if the frequency-sample index n is written as a binary number, the order is that binary number reversed. The bit-reversal process is illustrated here.

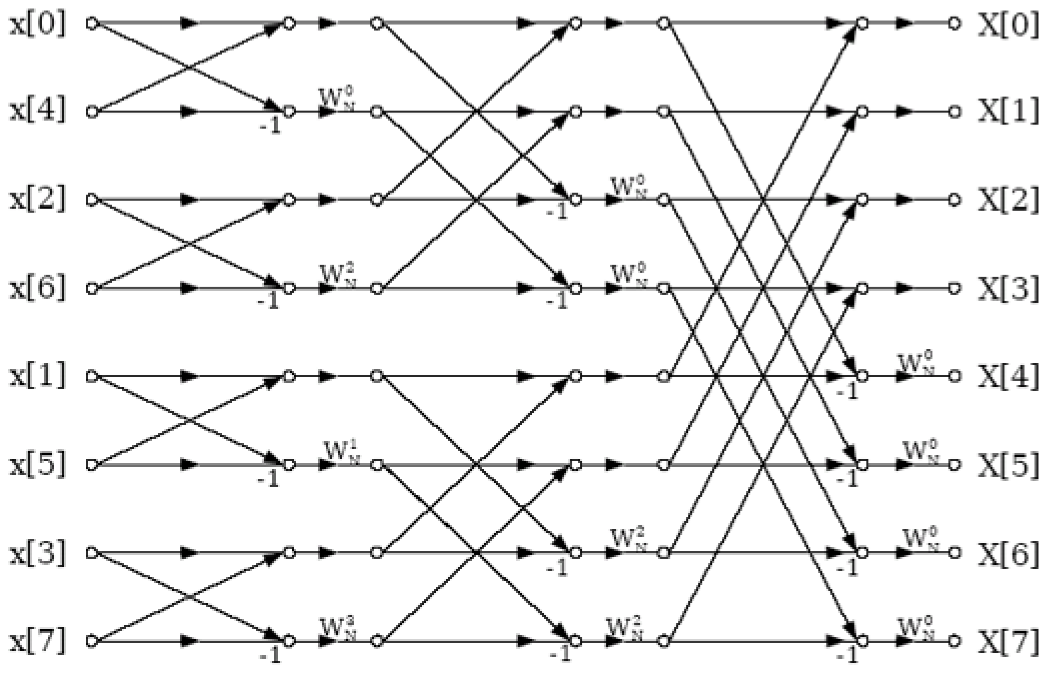
It is important to note that, if the input data are in order before beginning the FFT computations, the outputs of each butterfly throughout the computation can be placed in the same memory locations from which the inputs were fetched, resulting in an **in-place algorithm** that requires no extra memory to perform the FFT. Most FFT implementations are in-place, and overwrite the input data with the intermediate values and finally the output.

Alternate FFT Structures

Bit-reversing the input in decimation-in-time (DIT) FFTs or the output in decimation-in-frequency (DIF) FFTs can sometimes be inconvenient or inefficient. For such situations, alternate FFT structures have been developed. Such structures involve the same mathematical computations as the standard algorithms, but alter the memory locations in which intermediate values are stored or the order of computation of the FFT butterflies.

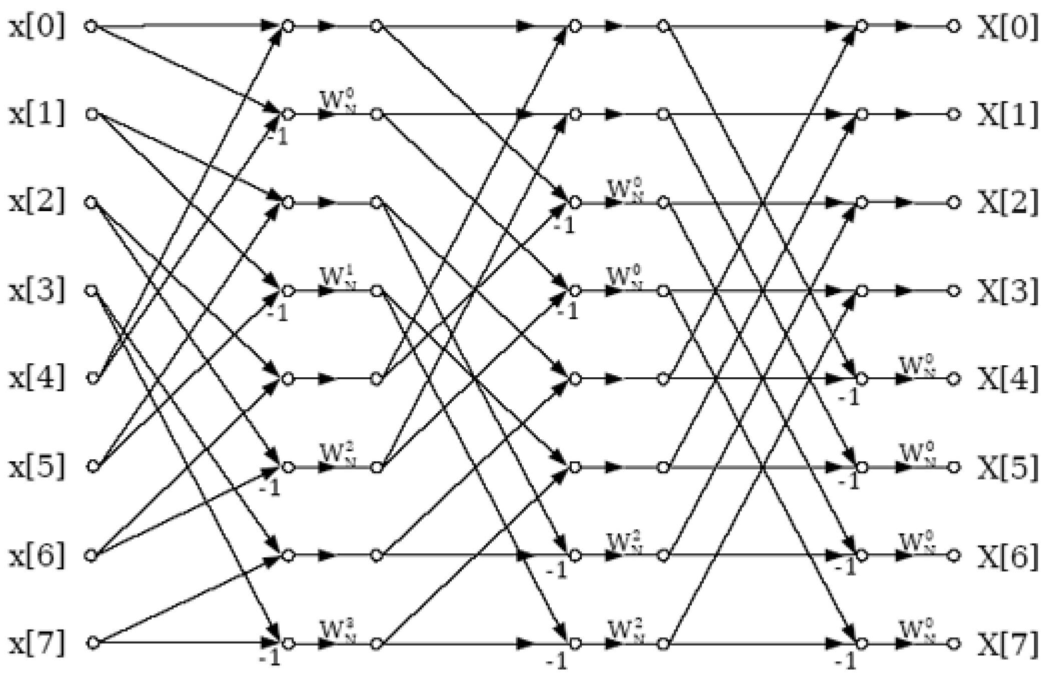
The structure in [link] computes a decimation-in-frequency FFT, but remaps the memory usage so that the **input** is bit-reversed, and the output is in-order as in the conventional decimation-in-time FFT. This alternate structure is still considered a DIF FFT because the twiddle factors are applied as in the DIF FFT. This structure is useful if for some reason the DIF butterfly is preferred but it is easier to bit-reverse the input.

Decimation-in-frequency radix-2 FFT with bit-reversed **input**. This is an in-place algorithm in which the same memory can be reused throughout the computation.



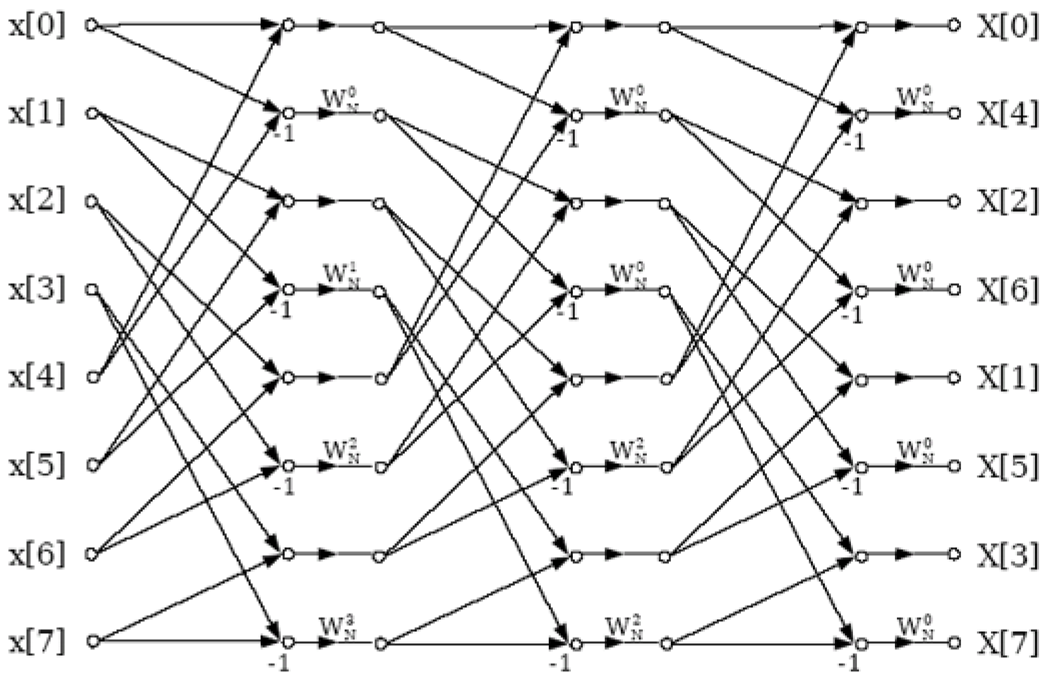
There is a similar structure for the decimation-in-time FFT with in-order inputs and bit-reversed frequencies. This structure can be useful for fast convolution on machines that favor decimation-in-time algorithms because the filter can be stored in bit-reverse order, and then the inverse FFT returns an in-order result without ever bit-reversing any data. As discussed in Efficient FFT Programming Tricks, this may save several percent of the execution time.

The structure in [link] implements a decimation-in-frequency FFT that has both input and output in order. It thus avoids the need for bit-reversing altogether. Unfortunately, it destroys the in-place structure somewhat, making an FFT program more complicated and requiring more memory; on most machines the resulting cost exceeds the benefits. This structure can be computed in place if **two** butterflies are computed simultaneously.



Decimation-in-frequency radix-2 FFT with in-order input and output. It can be computed in-place if two butterflies are computed simultaneously.

The structure in [link] has a constant geometry; the connections between memory locations are identical in each FFT stage. Since it is not in-place and requires bit-reversal, it is inconvenient for software implementation, but can be attractive for a highly parallel hardware implementation because the connections between stages can be hardwired. An analogous structure exists that has bit-reversed inputs and in-order outputs.



This constant-geometry structure has the same interconnect pattern from stage to stage. This structure is sometimes useful for special hardware.

Radix-4 FFT Algorithms

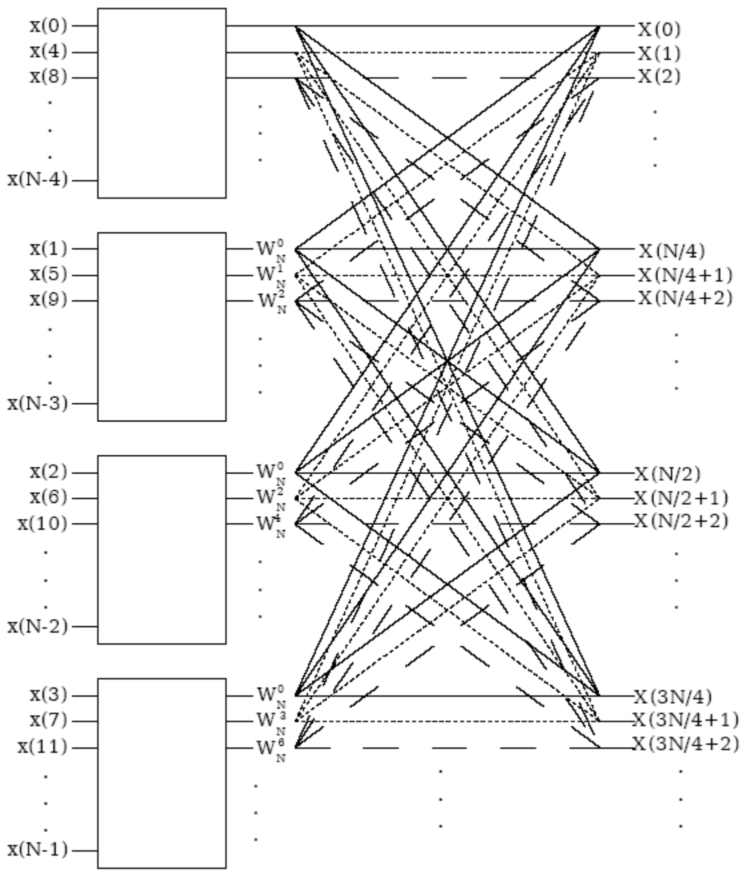
The radix-4 decimation-in-time and decimation-in-frequency fast Fourier transforms (FFTs) gain their speed by reusing the results of smaller, intermediate computations to compute multiple DFT frequency outputs. The radix-4 decimation-in-time algorithm rearranges the discrete Fourier transform (DFT) equation into four parts: sums over all groups of every fourth discrete-time index $n=[0,4,8,\ldots,N-4]$, $n=[1,5,9,\ldots,N-3]$, $n=[2,6,10,\ldots,N-2]$ and $n=[3,7,11,\ldots,N-1]$ as in [link]. (This works out only when the FFT length is a multiple of four.) Just as in the radix-2 decimation-in-time FFT, further mathematical manipulation shows that the length-N DFT can be computed as the sum of the outputs of four length-$\frac{N}{4}$ DFTs, of the even-indexed and odd-indexed discrete-time samples, respectively, where three of them are multiplied by so-called **twiddle factors** $W_N^k=e^{-\left(i\frac{2\pi k}{N}\right)}$, W_N^{2k}, and W_N^{3k}.

Equation:

$$\begin{aligned}X(k) &= \sum_{\mathrm{n}=0}^{N-1} x(n)e^{-\left(i\frac{2\pi nk}{N}\right)}\\ &= \sum_{n=0}^{\frac{N}{4}-1} x(4n)e^{-\left(i\frac{2\pi\times(4n)k}{N}\right)}+\sum_{n=0}^{\frac{N}{4}-1} x(4n+1)e^{-\left(i\frac{2\pi(4n+1)k}{N}\right)}+\sum_{n=0}^{\frac{N}{4}-1} x(4n+2)e^{-\left(i\frac{2\pi(4n+2)k}{N}\right)}+\sum_{n=0}^{\frac{N}{4}-}\\ &= \mathrm{DFT}_{\frac{N}{4}}[x(4n)]+W_N^k\,\mathrm{DFT}_{\frac{N}{4}}[x(4n+1)]+W_N^{2k}\,\mathrm{DFT}_{\frac{N}{4}}[x(4n+2)]+W_N^{3k}\,\mathrm{DFT}_{\frac{N}{4}}[x(4n+3)]\end{aligned}$$

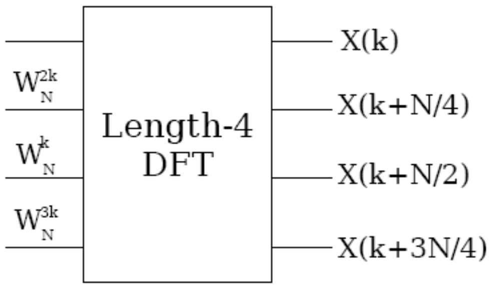
This is called a **decimation in time** because the time samples are rearranged in alternating groups, and a **radix-4** algorithm because there are four groups. [link] graphically illustrates this form of the DFT computation.

Radix-4 DIT structure



Decimation in time of a length-N DFT into four length-$\frac{N}{4}$ DFTs followed by a combining stage.

Due to the periodicity with $\frac{N}{4}$ of the short-length DFTs, their outputs for frequency-sample k are reused to compute $X(k)$, $X\left(k+\frac{N}{4}\right)$, $X\left(k+\frac{N}{2}\right)$, and $X\left(k+\frac{3N}{4}\right)$. It is this reuse that gives the radix-4 FFT its efficiency. The computations involved with each group of four frequency samples constitute the **radix-4 butterfly**, which is shown in [link]. Through further rearrangement, it can be shown that this computation can be simplified to three twiddle-factor multiplies and a length-4 DFT! The theory of multi-dimensional index maps shows that this must be the case, and that FFTs of any factorable length may consist of successive stages of shorter-length FFTs with twiddle-factor multiplications in between. The length-4 DFT requires no multiplies and only eight complex additions (this efficient computation can be derived using a radix-2 FFT).



[missing_resource: imageX.png]

The radix-4 DIT butterfly can be simplified to a length-4 DFT preceded by three twiddle-factor multiplies.

If the FFT length $N=4^M$, the shorter-length DFTs can be further decomposed recursively in the same manner to produce the full **radix-4 decimation-in-time FFT**. As in the radix-2 decimation-in-time FFT, each stage of decomposition creates additional savings in computation. To determine the total computational cost of the radix-4 FFT, note that there are $M=\log_4 N=\frac{\log_2 N}{2}$ stages, each with $\frac{N}{4}$ butterflies per stage. Each radix-4 butterfly requires 3 complex multiplies and 8 complex additions. The total cost is then

Radix-4 FFT Operation Counts

- $3\frac{N}{4}\frac{\log_2 N}{2}=\frac{3}{8}N\log_2 N$ complex multiplies (75% of a radix-2 FFT)
- $8\frac{N}{4}\frac{\log_2 N}{2}=N\log_2 N$ complex adds (same as a radix-2 FFT)

The radix-4 FFT requires only 75% as many complex multiplies as the radix-2 FFTs, although it uses the same number of complex additions. These additional savings make it a widely-used FFT algorithm.

The decimation-in-time operation regroups the input samples at each successive stage of decomposition, resulting in a "digit-reversed" input order. That is, if the time-sample index n is written as a base-4 number, the order is that base-4 number reversed. The digit-reversal process is illustrated for a length-$N=64$ example below.

Example:
N = 64 = 4^3

Original Number	Original Digit Order	Reversed Digit Order	Digit-Reversed Number
0	000	000	0
1	001	100	16
2	002	200	32
3	003	300	48
4	010	010	4
5	011	110	20
⋮	⋮	⋮	⋮

It is important to note that, if the input signal data are placed in digit-reversed order before beginning the FFT computations, the outputs of each butterfly throughout the computation can be placed in the same memory locations from which the inputs were fetched, resulting in an **in-place algorithm** that requires no extra memory to perform the FFT. Most FFT implementations are in-place, and overwrite the input data with the intermediate values and finally the output. A slight rearrangement within the radix-4 FFT introduced by Burrus allows the inputs to be arranged in bit-reversed rather than digit-reversed order.

A radix-4 decimation-in-frequency FFT can be derived similarly to the radix-2 DIF FFT, by separately computing all four groups of every fourth **output** frequency sample. The DIF radix-4 FFT is a flow-graph reversal of the DIT radix-4 FFT, with the same operation counts and twiddle factors in the reversed order. The output ends up in digit-reversed order for an in-place DIF algorithm.
Exercise:

Problem: How do we derive a radix-4 algorithm when $N = 4^M 2$?

Solution:

Perform a radix-2 decomposition for one stage, then radix-4 decompositions of all subsequent shorter-length DFTs.

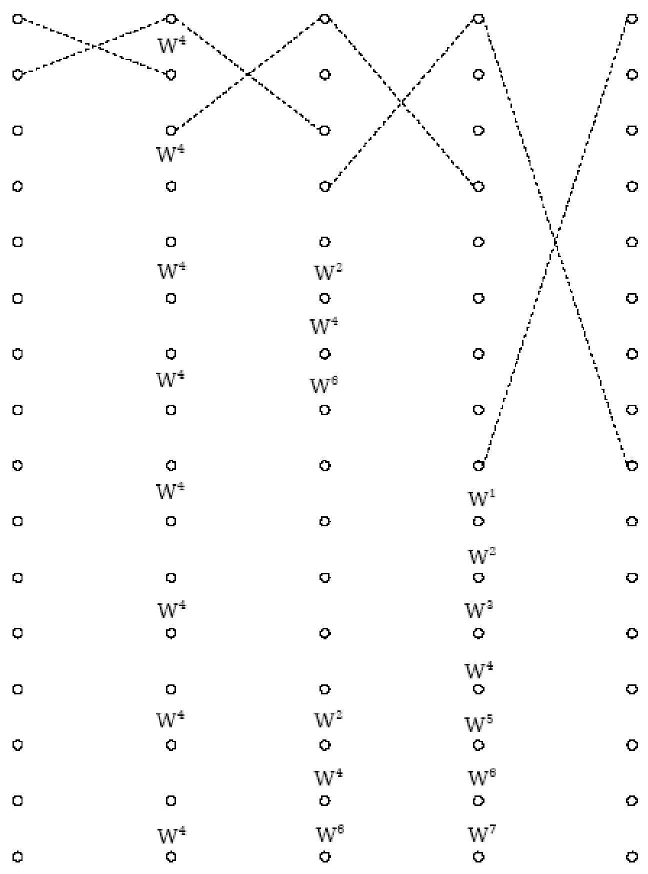
Split-radix FFT Algorithms

The split-radix algorithm, first clearly described and named by Duhamel and Hollman in 1984, required fewer total multiply and add operations operations than any previous power-of-two algorithm. (Yavne first derived essentially the same algorithm in 1968, but the description was so atypical that the work was largely neglected.) For a time many FFT experts thought it to be optimal in terms of total complexity, but even more efficient variations have more recently been discovered by Johnson and Frigo.

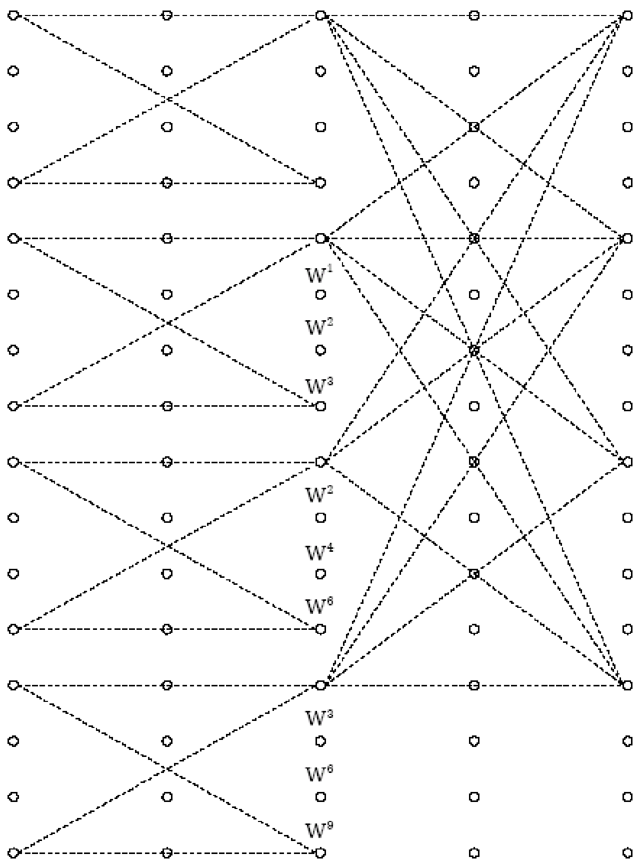
The split-radix algorithm can be derived by careful examination of the radix-2 and radix-4 flowgraphs as in Figure 1 below. While in most places the radix-4 algorithm has fewer nontrivial twiddle factors, in some places the radix-2 actually lacks twiddle factors present in the radix-4 structure or those twiddle factors simplify to multiplication by $-i$, which actually requires only additions. By mixing radix-2 and radix-4 computations appropriately, an algorithm of lower complexity than either can be derived.

Motivation for split-radix algorithm

radix-2

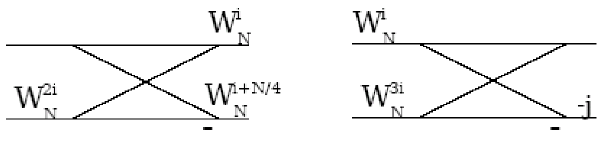


radix-4



See Decimation-in-Time (DIT) Radix-2 FFT and Radix-4 FFT Algorithms for more information on these algorithms.

An alternative derivation notes that radix-2 butterflies of the form shown in Figure 2 can merge twiddle factors from two successive stages to eliminate one-third of them; hence, the split-radix algorithm requires only about two-thirds as many multiplications as a radix-2 FFT.



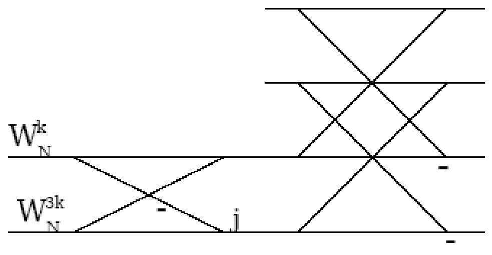
Note that these two butterflies are equivalent

The split-radix algorithm can also be derived by mixing the radix-2 and radix-4 decompositions.
Equation:

DIT Split-radix derivation

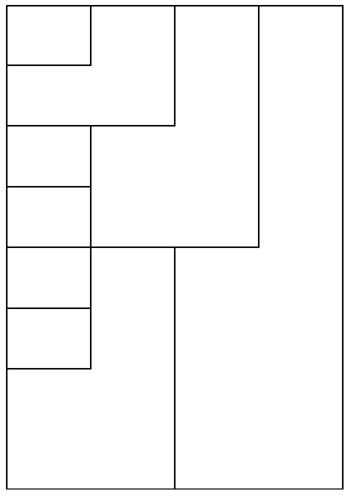
$$\begin{aligned} X(k) &= \sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-\left(i\frac{2\pi\times(2n)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+1)e^{-\left(i\frac{2\pi(4n+1)k}{N}\right)} + \sum_{n=0}^{\frac{N}{4}-1} x(4n+3)e^{-\left(i\frac{2\pi(4n+3)k}{N}\right)} \\ &= \mathrm{DFT}_{\frac{N}{2}}[x(2n)] + W_N^k \mathrm{DFT}_{\frac{N}{4}}(x(4n+1)) + W_N^{3k}\mathrm{DFT}_{\frac{N}{4}}(x(4n+3)) \end{aligned}$$

Figure 3 illustrates the resulting split-radix butterfly.
Decimation-in-Time Split-Radix Butterfly



The split-radix butterfly mixes radix-2 and radix-4 decompositions and is L-shaped

Further decomposition of the half- and quarter-length DFTs yields the full split-radix algorithm. The mix of different-length FFTs in different parts of the flowgraph results in a somewhat irregular algorithm; Sorensen et al. show how to adjust the computation such that the data retains the simpler radix-2 bit-reverse order. A decimation-in-frequency split-radix FFT can be derived analogously.



The split-radix transform has L-shaped butterflies

The multiplicative complexity of the split-radix algorithm is only about two-thirds that of the radix-2 FFT, and is better than the radix-4 FFT or any higher power-of-two radix as well. The additions within the complex twiddle-factor multiplies are similarly reduced, but since the underlying butterfly tree remains the same in all power-of-two algorithms, the butterfly additions remain the same and the overall reduction in additions is much less.

	Complex M/As	**Real M/As (4/2)**	**Real M/As (3/3)**
Multiplies	$O\left[\frac{N}{3}\log_2 N\right]$	$\frac{4}{3}N\log_2 N - \frac{38}{9}N + 6 + \frac{2}{9}(-1)^{\mathrm{M}}$	$N\log_2 N - 3N + 4$
Additions	$O[N\log_2 N]$	$\frac{8}{3}N\log_2 N - \frac{16}{9}N + 2 + \frac{2}{9}(-1)^{\mathrm{M}}$	$3N\log_2 N - 3N + 4$

Operation Counts

Comments

- The split-radix algorithm has a somewhat irregular structure. Successful progams have been written (Sorensen) for uni-processor machines, but it may be difficult to efficiently code the split-radix algorithm for vector or multi-processor machines.
- G. Bruun's algorithm requires only $N - 2$ more operations than the split-radix algorithm and has a regular structure, so it might be better for multi-processor or special-purpose hardware.
- The execution time of FFT programs generally depends more on compiler- or hardware-friendly software design than on the exact computational complexity. See Efficient FFT Algorithm and Programming Tricks for further pointers and links to good code.

Multidimensional Index Maps for DIF and DIT algorithms

Decimation-in-time algorithm

Radix-2 DIT:

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk}=\sum_{n=0}^{\frac{N}{2}-1}x(2n)W_N^{2nk}+\sum_{n=0}^{\frac{N}{2}-1}x(2n+1)W_N^{(2n+1)k}$$

Formalization: Let $n=n_1+2n_2$: $n_1=[0,1]$: $n_2=\left[0,1,2,\dots,\frac{N}{2}-1\right]$

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk}=\sum_{n_1=0}^{1}\sum_{n_2=0}^{\frac{N}{2}-1}x(n_1+2n_2)W_N^{(n_1+2n_2)k}$$

Also, let $k=\frac{N}{2}k_1+k_2$: $k_1=[0,1]$: $k_2=\left[0,1,2,\dots,\frac{N}{2}-1\right]$

Note: As long as there is a one-to-one correspondence between the original indices $[n,k]=[0,1,2,\dots,N-1]$ and the n, k generated by the index map, the computation is the **same**; only the order in which the sums are done is changed.

Rewriting the DFT formula in terms of index map $n=n_1+2n_2$, $k=\frac{N}{2}k_1+k_2$:

Equation:

Math input error

Note: Key to FFT is choosing index map so that one of the cross-terms disappears!

Exercise:

Problem: What is an index map for a radix-4 DIT algorithm?

Exercise:

Problem: What is an index map for a radix-4 DIF algortihm?

Exercise:

Problem:

What is an index map for a radix-3 DIT algorithm? (N a multiple of 3)

For arbitrary composite N $= N_1 N_2$, we can define an index map

$$n = n_1 + N_1 n_2$$

$$k = N_2 k_1 + k_2$$

$$n_1 = [0, 1, 2, \ldots, N_1 - 1]$$

$$k_1 = [0, 1, 2, \ldots, N_1 - 1]$$

$$n_2 = [0, 1, 2, \ldots, N_2 - 1]$$

$$k_2 = [0, 1, 2, \ldots, N_2 - 1]$$

Equation:

$$\begin{aligned}
X(k) &= X(k_1, k_2) \\
&= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_N^{N_2 n_1 k_1} W_N^{n_1 k_2} W_N^{N k_1 n_2} W_N^{N_1 n_2 k_2} \\
&= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_{N_1}^{n_1 k_1} W_N^{n_1 k_2} 1 W_{N_2}^{n_2 k_2} \\
&= \mathrm{DFT}_{n_1, N_1} \left[W_N^{n_1 k_2} \mathrm{DFT}_{n_2, N_2} \left[x(n_1, n_2) \right] \right]
\end{aligned}$$

Computational cost in multipliesl "Common Factor Algorithm (CFA)"

- N_1 length-N_2 DFTs ⇒ $N_1 {N_2}^2$
- N twiddle factors ⇒ N
- N_2 length-N_1 DFTs ⇒ $N_2 {N_1}^2$
- **Total** $N_1 {N_2}^2 + N_1 N_2 + N_2 {N_1}^2 = N \left(N_1 + N_2 + 1\right)$

"Direct": $N^2 = N \left(N_1 N_2\right)$

Example:

$$N_1 = 16$$

$$N_2 = 15$$

$$N = 240$$

$$\text{direct} = 240^2 = 57600$$

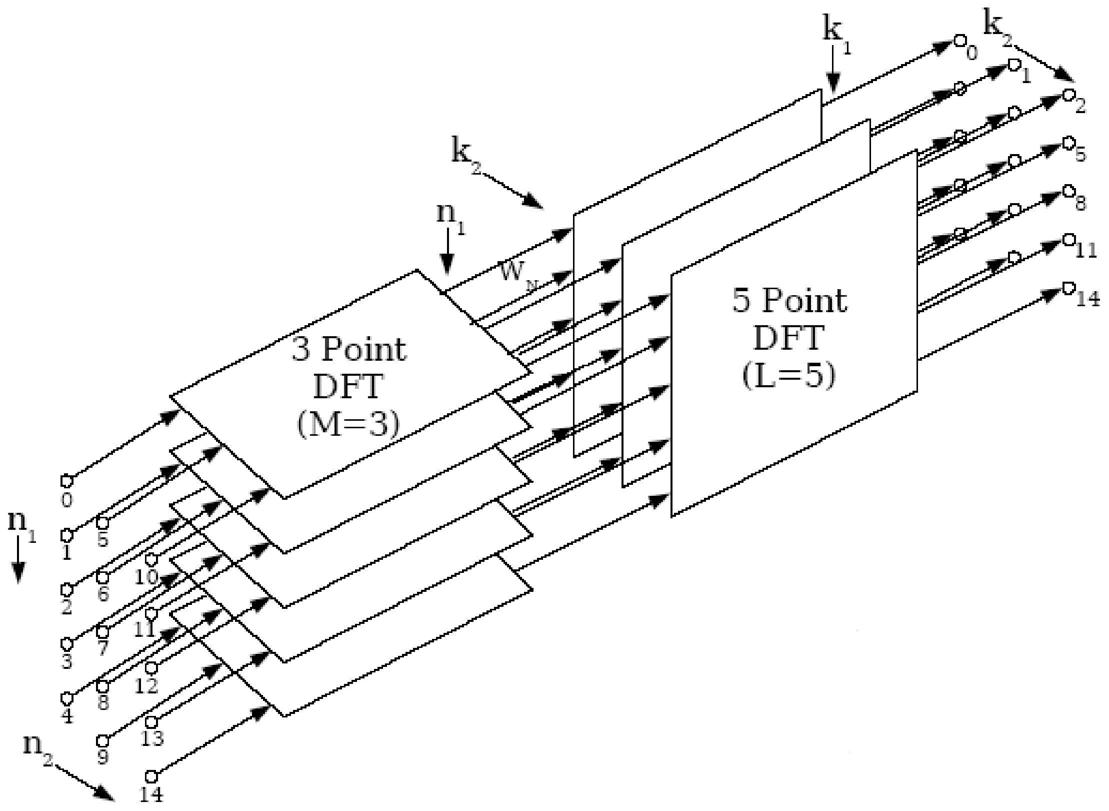
$$\text{CFA} = 7680$$

Tremendous saving for **any** composite N

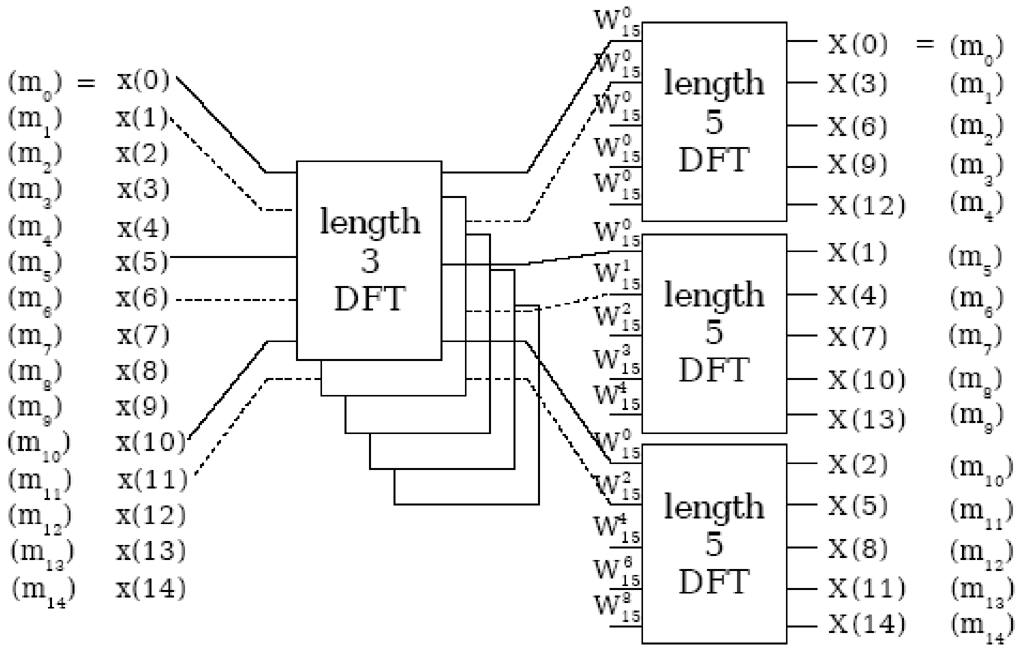
Pictorial Representations

$$n = n_1 + 5 n_2, \; k = 3 k_1 + k_2$$

Emphasizes Multi-dimensional structure

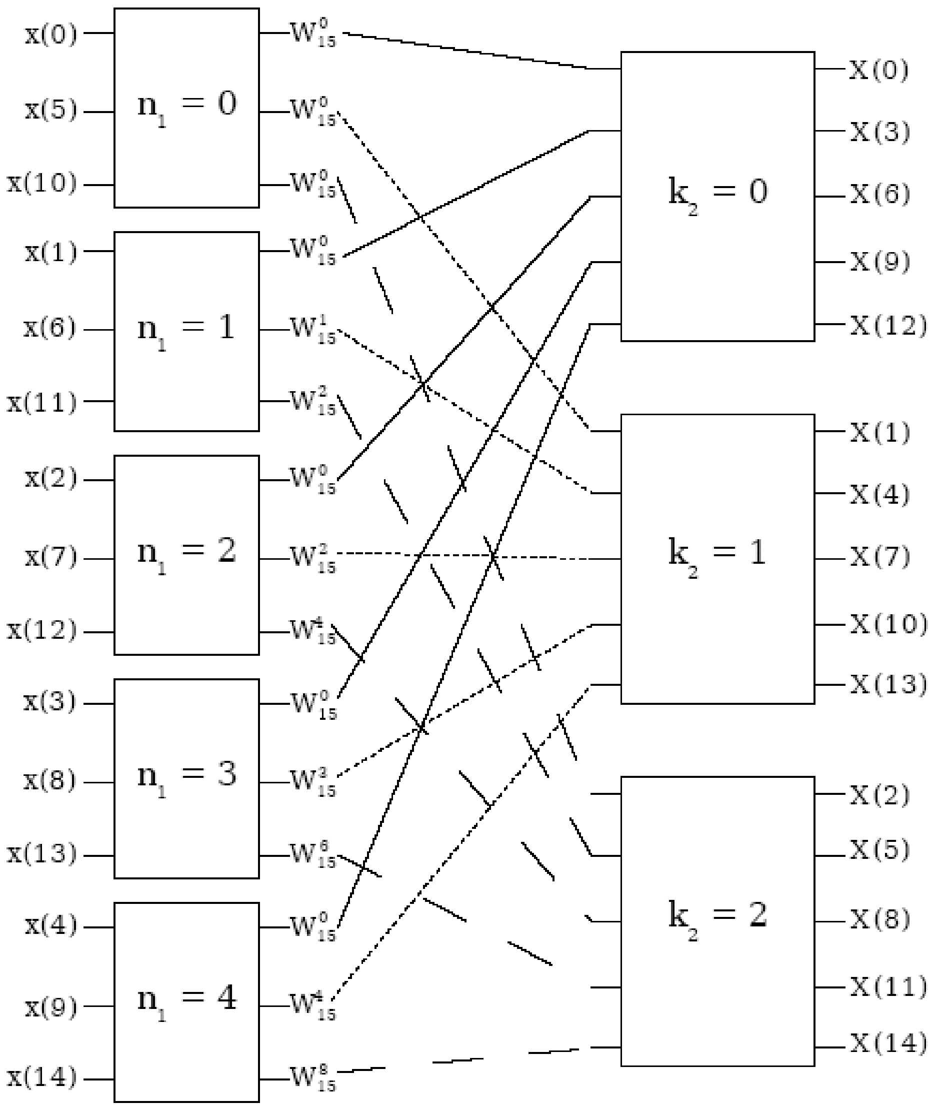


Emphasizes computer memory organization



(m$_0$) = x(0)
(m$_1$) x(1)
(m$_2$) x(2)
(m$_3$) x(3)
(m$_4$) x(4)
(m$_5$) x(5)
(m$_6$) x(6)
(m$_7$) x(7)
(m$_8$) x(8)
(m$_9$) x(9)
(m$_{10}$) x(10)
(m$_{11}$) x(11)
(m$_{12}$) x(12)
(m$_{13}$) x(13)
(m$_{14}$) x(14)
length 3 DFT
W_{15}^0 W_{15}^0 W_{15}^0 W_{15}^0 W_{15}^0
length 5 DFT
W_{15}^0 W_{15}^1 W_{15}^2 W_{15}^3 W_{15}^4
length 5 DFT
W_{15}^0 W_{15}^2 W_{15}^4 W_{15}^6 W_{15}^8
length 5 DFT
X(0) = (m$_0$)
X(3) (m$_1$)
X(6) (m$_2$)
X(9) (m$_3$)
X(12) (m$_4$)
X(1) (m$_5$)
X(4) (m$_6$)
X(7) (m$_7$)
X(10) (m$_8$)
X(13) (m$_9$)
X(2) (m$_{10}$)
X(5) (m$_{11}$)
X(8) (m$_{12}$)
X(11) (m$_{13}$)
X(14) (m$_{14}$)

Easy to draw



Exercise:

Problem: Can the composite CFAs be implemented in-place?

Exercise:

Problem: What do we do with $N = N_1 N_2 N_3$?

The Prime Factor Algorithm

General Index Maps

$$n = (K_1 n_1 + K_2 n_2) \bmod N$$

$$n = (K_3 k_1 + K_4 k_2) \bmod N$$

$$n_1 = [0, 1, \ldots, N_1 - 1]$$

$$k_1 = [0, 1, \ldots, N_1 - 1]$$

$$n_2 = [0, 1, \ldots, N_2 - 1]$$

$$k_2 = [0, 1, \ldots, N_2 - 1]$$

The basic ideas is to simply **reorder** the DFT computation to expose the redundancies in the DFT, and exploit these to reduce computation!

Three conditions must be satisfied to make this map serve our purposes

1. Each map must be one-to-one from 0 to $N-1$, because we want to do the **same** computation, just in a different order.
2. The map must be cleverly chosen so that computation is reduced
3. The map should be chosen to make the short-length transforms be DFTs. (Not essential, since fast algorithms for short-length DFT-like computations could be developed, but it makes our work easier.)

Conditions for one-to-oneness of general index map

Case I

N_1, N_2 relatively prime (greatest common denominator 1) i.e. gcd $(N_1, N_2) = 1$

$K_1 = aN_2$ and/or $K_2 = bN_1$ and gcd $(K_1, N_1) = 1$, gcd $(K_2, N_2) = 1$

Case II

N_1, N_2 **not** relatively prime: gcd $(N_1, N_2) > 1$

$K_1 = aN_2$ and $K_2 \neq bN_1$ and gcd $(a, N_1) = 1$, gcd $(K_2, N_2) = 1$ or $K_1 \neq aN_2$ and $K_2 = bN_1$ and gcd $(K_1, N_1) = 1$, gcd $(b, N_2) = 1$ where K_1, K_2, K_3, K_4, N_1, N_2, a, b integers

Note: Requires number-theory/abstract-algebra concepts. Reference: C.S. Burrus

Note: Conditions of one-to-oneness must apply to both k and n

Conditions for arithmetic savings

Equation:

$$\begin{aligned} X(k_1, k_2) &= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_N^{(K_1 n_1 + K_2 n_2)(K_3 k_1 + K_4 k_2)} \\ &= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_N^{K_1 K_3 n_1 k_1} W_N^{K_1 K_4 n_1 k_2} W_N^{K_2 K_3 n_2 k_1} W_N^{K_2 K_4 n_2 k_2} \end{aligned}$$

- $K_1 K_4 \bmod N = 0$ exclusive or $K_2 K_3 \bmod N = 0$ ⇒ Common Factor Algorithm (CFA). Then

  $$X(k) = \mathrm{DFT}_{Ni}\left[\text{twiddle factors}\,\mathrm{DFT}_{Nj}\left[x(n_1, n_2)\right]\right]$$

- $K_1 K_4 \bmod N$ **and** $K_2 K_3 \bmod N = 0$ ⇒ Prime Factor Algorithm (PFA).

  $$X(k) = \mathrm{DFT}_{Ni}\left[\mathrm{DFT}_{Nj}\right]$$

  No twiddle factors!

Note: A PFA exists only and always for relatively prime N_1, N_2

Conditions for short-length transforms to be DFTs

$K_1 K_3 \bmod N = N_2$ and $K_2 K_4 \bmod N = N_1$

Note: Convenient choice giving a PFA

$K_1 = N_2$, $K_2 = N_1$, $K_3 = N_2{N_2}^{-1} \bmod N_1 \bmod N_1$,
$K_4 = N_1{N_1}^{-1} \bmod N_2 \bmod N_2$ where ${N_1}^{-1} \bmod N_2$ is an integer such that $N_1{N_1}^{-1} \text{ mod} = 1$

Example:
$N_1 = 3$, $N_2 = 5$ $N = 15$

$$\text{n} = (5n_1 + 3n_2) \bmod 15$$

$$k = (10k_1 + 6k_2) \bmod 15$$

1. **Checking Conditions for one-to-oneness**

$$5 = K_1 = aN_2 = 5a$$

$$3 = K_2 = bN_1 = 3b$$

$$\gcd(5, 3) = 1$$

$$\gcd(3, 5) = 1$$

$$10 = K_3 = aN_2 = 5a$$

$$6 = K_4 = bN_1 = 3b$$

$$\gcd(10, 3) = 1$$

$$\gcd(6, 5) = 1$$

2. **Checking conditions for reduced computation**

$$K_1K_4 \bmod 15 = 5 \times 6 \bmod 15 = 0$$

$$K_2K_3 \bmod 15 = 3 \times 10 \bmod 15 = 0$$

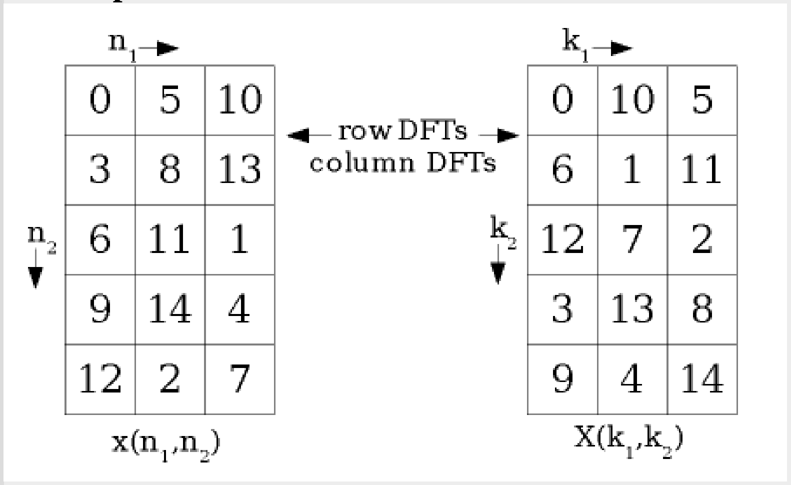
3. **Checking Conditions for making the short-length transforms be DFTS**

$$K_1K_3 \bmod 15 = 5 \times 10 \bmod 15 = 5 = N_2$$

$$K_2K_4 \bmod 15 = 3 \times 6 \bmod 15 = 3 = N_1$$

Therefore, this is a prime factor map.

2-D map



$n = (5n_1 + 3n_2) \bmod 15$ and
$k = (10k_1 + 6k_2) \bmod 15$

Operation Counts

- N_2 length- N_1 DFTs N_1 length- N_2 DFTs

  $$N_2{N_1}^2 + N_1{N_2}^2 = N\left(N_1 + N_2\right)$$

  complex multiplies
- Suppose $N = N_1 N_2 N_3 \dots N_{\mathrm{M}}$

  $$N\left(N_1 + N_2 + \dots + N_M\right)$$

  Complex multiplies

Note: radix-2, radix-4 eliminate all multiplies in short-length DFTs, but have twiddle factors: PFA eliminates all twiddle factors, but ends up with multiplies in short-length DFTs. Surprisingly, total operation counts end up being very similar for similar lengths.

Fast Convolution

Fast Circular Convolution

Since,

$$\sum_{m=0}^{N-1} x(m)h(n-m) \bmod N = y(n) \text{is equivalent to} Y(k) = X(k)H(k)$$

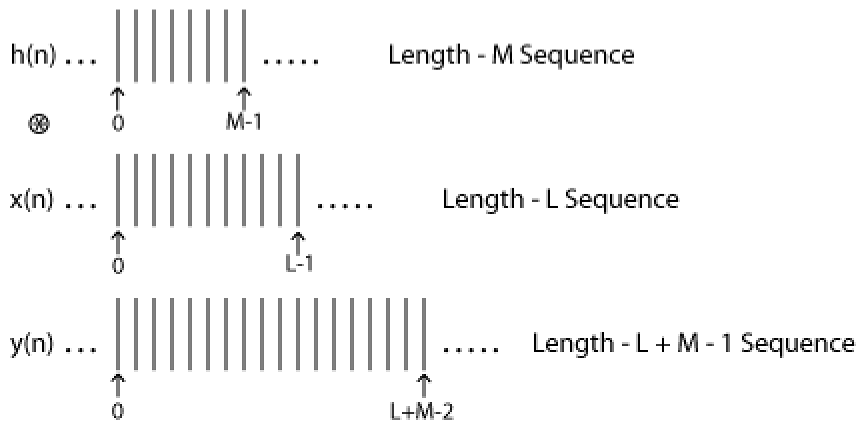
$y(n)$ can be computed as $y(n) = \mathrm{IDFT}\left[\mathrm{DFT}\left[x(n)\right]\mathrm{DFT}\left[h(n)\right]\right]$
Cost

- **Direct**
  - N^2 complex multiplies.
  - $N(N-1)$ complex adds.
- **Via FFTs**
  - 3 FFTs + N multipies.
  - $N + \frac{3N}{2}\log_2 N$ complex multiplies.
  - $3(N\log_2 N)$ complex adds.

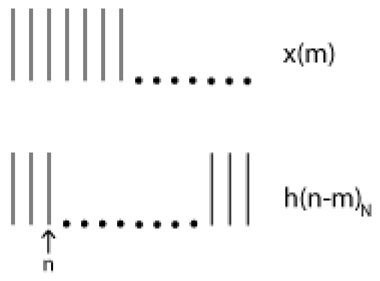
If $H(k)$ can be precomputed, cost is only 2 FFts + N multiplies.

Fast Linear Convolution

DFT produces cicular convolution. For linear convolution, we must zero-pad sequences so that circular wrap-around always wraps over zeros.



To achieve linear convolution using fast circular convolution, we must use zero-padded DFTs of length $N \geq L + M - 1$



Choose shortest convenient N (usually smallest power-of-two greater than or equal to $L + M - 1$)

$$y(n) = \mathrm{IDFT}_N\left[\mathrm{DFT}_N\left[x(n)\right]\mathrm{DFT}_N\left[h(n)\right]\right]$$

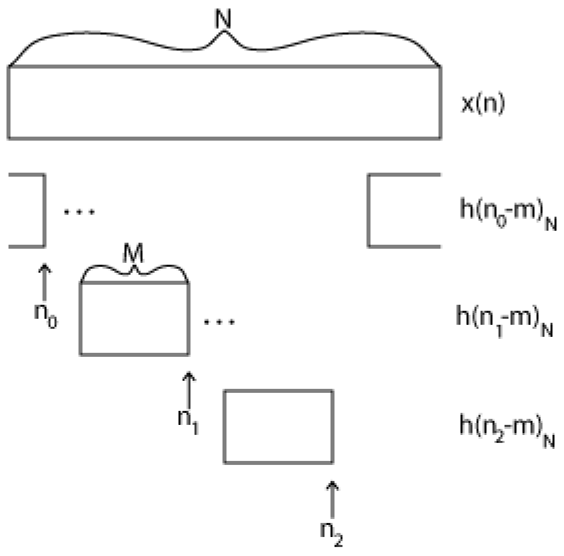
Note: There is some inefficiency when compared to circular convolution due to longer zero-padded DFTs. Still, $O\left(\frac{N}{\log_2 N}\right)$ savings over direct computation.

Running Convolution

Suppose $L = \infty$, as in a real time filter application, or $L \gg M$. There are efficient block methods for computing fast convolution.

Overlap-Save (OLS) Method

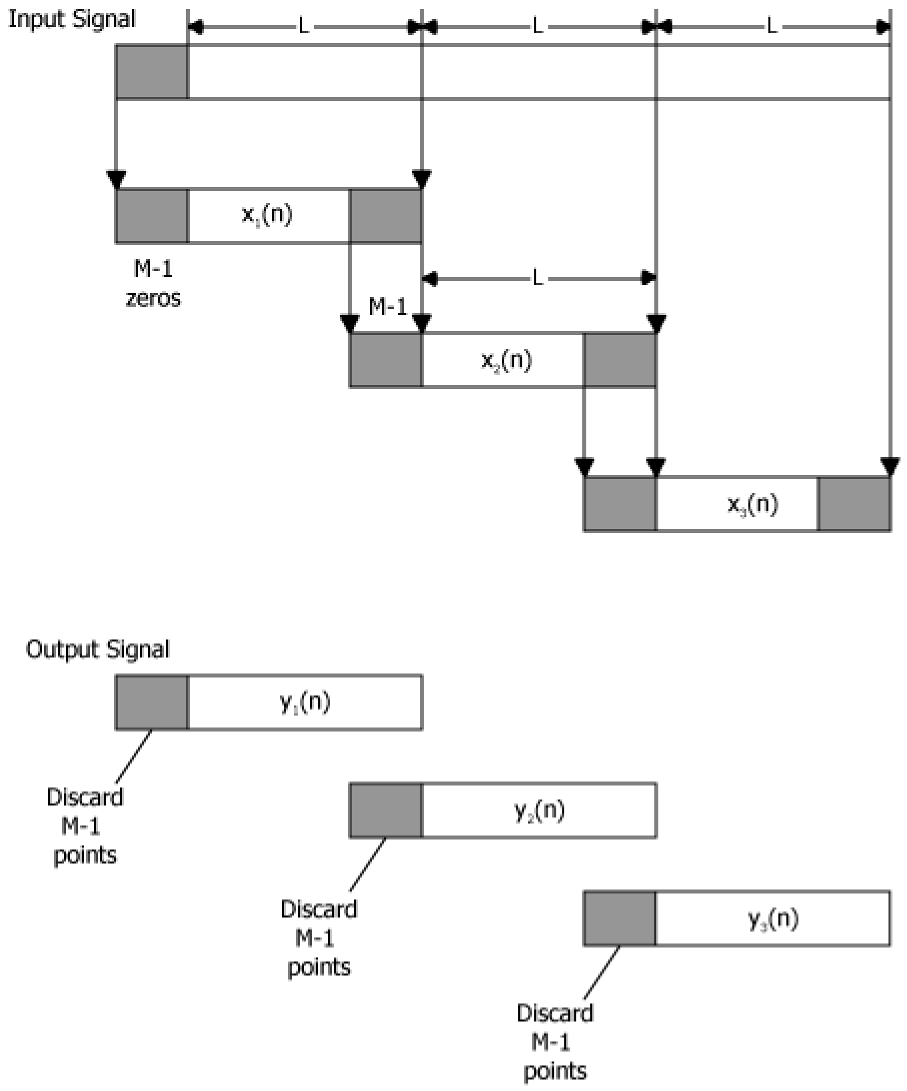
Note that if a length-M filter $h(n)$ is circularly convulved with a length-N segment of a signal $x(n)$,



the first $M-1$ samples are wrapped around and thus is **incorrect**. However, for $M-1 \leq n \leq N-1$,the convolution is linear convolution, so these samples are correct. Thus $N-M+1$ good outputs are produced for each length-N circular convolution.

The Overlap-Save Method: Break long signal into successive blocks of N samples, each block overlapping the previous block by $M-1$ samples. Perform circular convolution of each block with filter $h(m)$. Discard first

$M-1$ points in each output block, and concatenate the remaining points to create $y(n)$.



Computation cost for a length-N equals 2^n FFT per output sample is (assuming precomputed $H(k)$) 2 FFTs and N multiplies

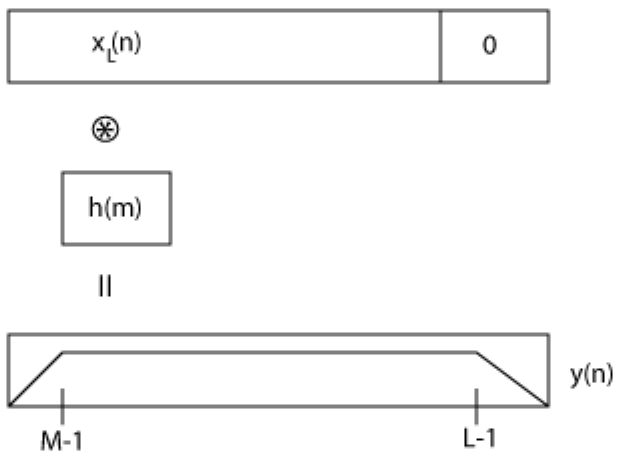
$$\frac{2\left(\frac{N}{2}\log_2 N\right)+N}{N-M+1}=\frac{N\left(\log_2 N+1\right)}{N-M+1}\text{complex multiplies}$$

$$\frac{2\left(N\log_2 N\right)}{N-M+1} = \frac{2N\log_2 N}{N-M+1}\text{complex adds}$$

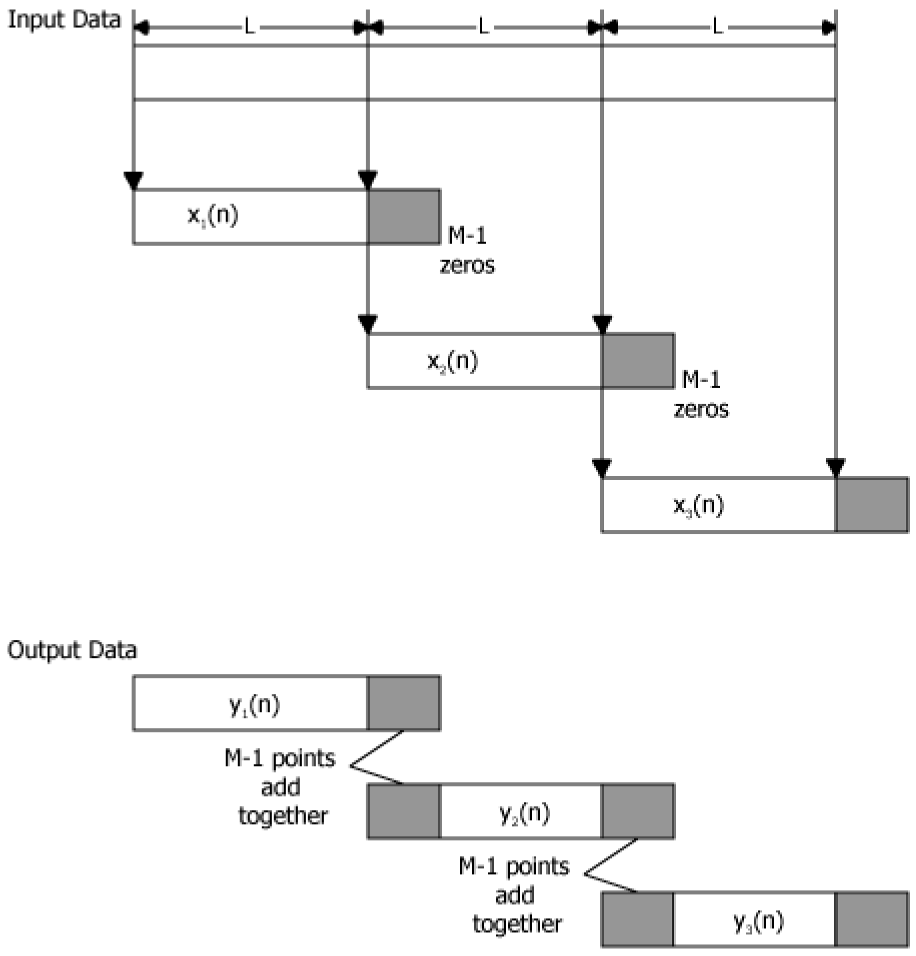
Compare to M mults, $M-1$ adds per output point for direct method. For a given M, optimal N can be determined by finding N minimizing operation counts. Usualy, optimal N is $4M \le N_{\text{opt}} \le 8M$.

Overlap-Add (OLA) Method

Zero-pad length-L blocks by $M-1$ samples.



Add successive blocks, overlapped by $M-1$ samples, so that the tails sum to produce the complete linear convolution.



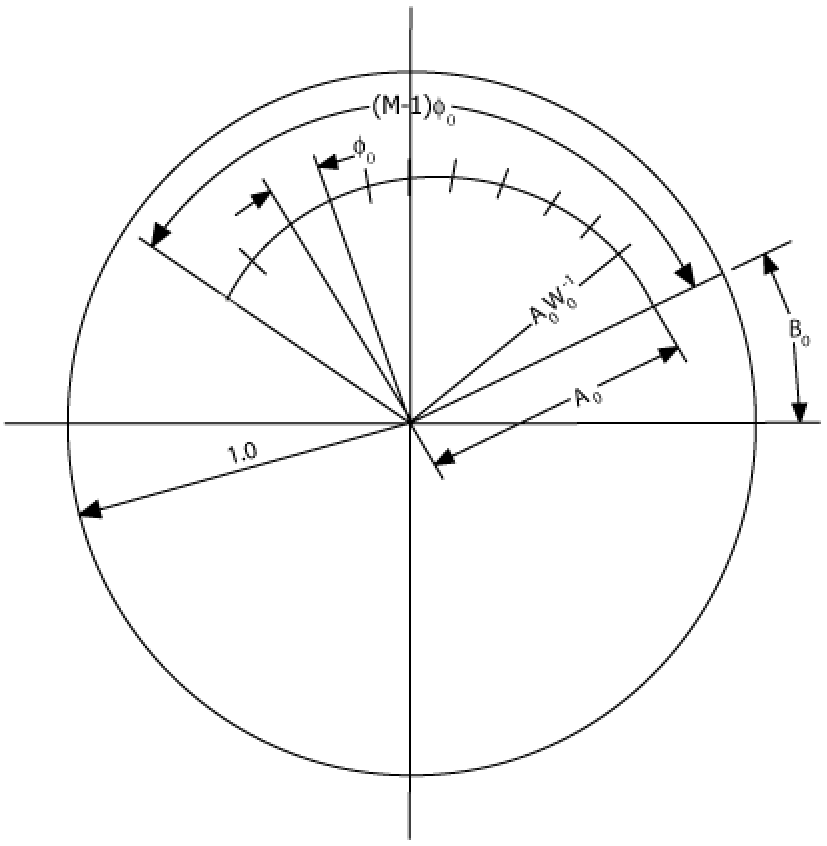
Computational Cost: Two length $N = L + M - 1$ FFTs and M mults and $M - 1$ adds per L output points; essentially the sames as OLS method.

Chirp-z Transform

Let $z^k = AW^{-k}$, where $A = A_o e^{i\theta_o}$, $W = W_o e^{-(i\varphi_o)}$.

We wish to compute M samples, $k = [0, 1, 2, \ldots, M-1]$ of

$$X(z_k) = \sum_{n=0}^{N-1} x(n) {z_k}^{-n} = \sum_{n=0}^{N-1} x(n) A^{-n} W^{\mathrm{nk}}$$



Note that
$\left((k-n)^2 = n^2 - 2nk + k^2\right) \Rightarrow \left(nk = \frac{1}{2}\left(n^2 + k^2 - (k-n)^2\right)\right)$, So

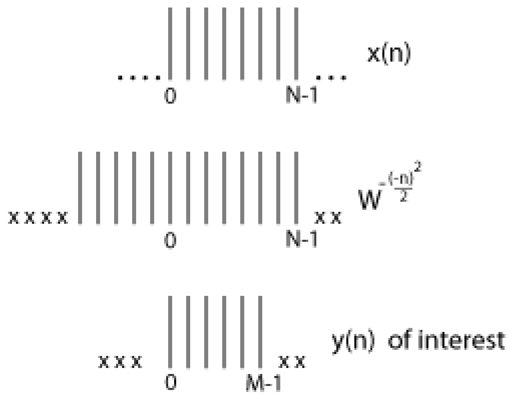
$$X(z_k) = \sum_{n=0}^{N-1} x(n) A^{-n} W^{\frac{n^2}{2}} W^{\frac{k^2}{2}} W^{\frac{-(k-n)^2}{2}}$$

$$W^{\frac{k^2}{2}} \sum_{n=0}^{N-1} x(n) A^{-n} W^{\frac{n^2}{2}} W^{\frac{-(k-n)^2}{2}}$$

Thus, $X(z_k)$ can be compared by

1. Premultiply $x(n)$ by $A^n W^{\frac{n^2}{2}}$, $n = [0, 1, \ldots, N-1]$ to make $y(n)$
2. Linearly convolve with $W^{\frac{-(k-n)^2}{2}}$
3. Post multiply by to get $W^{\frac{k^2}{2}}$ to get $X(z_k)$.

1. and 3. require N and M operations respectively. 2. can be performed efficiently using fast convolution.



$W^{-\frac{n^2}{2}}$ is required only for $-(N-1) \leq n \leq M-1$, so this linear convolution can be implemented with $L \geq N + M - 1$ FFTs.

Note: Wrap $W^{-\frac{n^2}{2}}$ around L when implementing with circular convolution.

So, a weird-length DFT can be implemented relatively efficiently using power-of-two algorithms via the chirp-z transform.

Also useful for "zoom-FFTs".

FFTs of prime length and Rader's conversion

The power-of-two FFT algorithms, such as the radix-2 and radix-4 FFTs, and the common-factor and prime-factor FFTs, achieve great reductions in computational complexity of the DFT when the length, N, is a composite number. DFTs of prime length are sometimes needed, however, particularly for the short-length DFTs in common-factor or prime-factor algorithms. The methods described here, along with the composite-length algorithms, allow fast computation of DFTs of **any** length.

There are two main ways of performing DFTs of prime length:

1. Rader's conversion, which is most efficient, and the
2. Chirp-z transform, which is simpler and more general.

Oddly enough, both work by turning prime-length DFTs into convolution! The resulting convolutions can then be computed efficiently by either

1. fast convolution via composite-length FFTs (simpler) or by
2. Winograd techniques (more efficient)

Rader's Conversion

Rader's conversion is a one-dimensional index-mapping scheme that turns a length-N DFT (N prime) into a length-($N-1$) convolution and a few additions. Rader's conversion works **only** for prime-length N.

An **index map** simply rearranges the order of the sum operation in the DFT definition. Because addition is a commutative operation, the same mathematical result is produced from any order, as long as all of the same terms are added once and only once. (This is the condition that defines an index map.) Unlike the multi-dimensional index maps used in deriving common factor and prime-factor FFTs, Rader's conversion uses a one-dimensional index map in a finite group of N integers: $k = r^m \bmod N$

Fact from number theory

If N is prime, there exists an integer "r" called a **primitive root**, such that the index map $k = r^m \bmod N, m = [0, 1, 2, \ldots, N-2]$, uniquely generates all elements $k = [1, 2, 3, \ldots, N-1]$

Example:
$N = 5, r = 2$

$$2^0 \bmod 5 = 1$$

$$2^1 \bmod 5 = 2$$

$$2^2 \bmod 5 = 4$$

$$2^3 \bmod 5 = 3$$

Another fact from number theory

For N prime, the inverse of r (i.e. $r^{-1} r \bmod N = 1$ is also a primitive root (call it r^{-1}).

Example:
$N = 5, r = 2 \; r^{-1} = 3$

$$2 \times 3 \bmod 5 = 1$$

$$3^0 \bmod 5 = 1$$

$$3^1 \bmod 5 = 3$$

$$3^2 \bmod 5 = 4$$

$$3^3 \bmod 5 = 2$$

So why do we care? Because we can use these facts to turn a DFT into a convolution!

Rader's Conversion

Let
$\forall mn, (m = [0, 1, ..., N-2]) \ \wedge \ n \in [1, 2, ..., N-1] : (n = r^{-m} \bmod N)$
$, \forall pk, (p = [0, 1, ..., N-2]) \ \wedge \ k \in [1, 2, ..., N-1] : (k = r^{p} \bmod N)$

$$X(k) = \sum_{n=0}^{N-1} x(\text{n}) W_N^{nk} = \begin{cases} x(0) + \sum_{n=1}^{N-1} x(n) W_N^{nk} \ \text{ if } \ k \neq 0 \\ \sum_{n=0}^{N-1} x(n) \ \text{ if } \ k = 0 \end{cases}$$

where for convenience $W_N^{nk} = e^{-\left(i\frac{2\pi nk}{N}\right)}$ in the DFT equation. For $k \neq 0$
Equation:

$$\begin{aligned} X(r^p \bmod N) &= \sum_{m=0}^{N-2} x(r^{-m} \bmod N) W^{r^p r^{-m}} + x(0) \\ &= \sum_{m=0}^{N-2} x(r^{-m} \bmod N) W^{r^{p-m}} + x(0) \\ &= x(0) + x\left(r^{-l} \bmod N\right) * W^{r^l} \end{aligned}$$

where $l = [0, 1, ..., N-2]$

Example:
$N = 5, r = 2, r^{-1} = 3$

$$\begin{pmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ X(4) \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 2 & 4 & 1 & 3 \\ 0 & 3 & 1 & 4 & 2 \\ 0 & 4 & 3 & 2 & 1 \end{pmatrix} \begin{pmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \end{pmatrix}$$

$$\begin{pmatrix} X(0) \\ X(1) \\ X(2) \\ X(4) \\ X(3) \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 3 & 4 & 2 \\ 0 & 2 & 1 & 3 & 4 \\ 0 & 4 & 2 & 1 & 1 \\ 0 & 3 & 4 & 2 & 3 \end{pmatrix} \begin{pmatrix} x(0) \\ x(1) \\ x(3) \\ x(4) \\ x(2) \end{pmatrix}$$

where for visibility the matrix entries represent only the **power**, m of the corresponding DFT term W_N^m Note that the 4-by-4 circulant matrix

$$\begin{pmatrix} 1 & 3 & 4 & 2 \\ 2 & 1 & 3 & 4 \\ 4 & 2 & 1 & 1 \\ 3 & 4 & 2 & 3 \end{pmatrix}$$

corresponds to a length-4 circular convolution.

Rader's conversion turns a prime-length DFT into a few adds and a **composite-length** $(N-1)$ circular convolution, which can be computed efficiently using either

1. fast convolution via FFT and IFFT
2. index-mapped convolution algorithms and short Winograd convolution alogrithms. (Rather complicated, and trades fewer multiplies for many more adds, which may not be worthwile on most modern processors.) See R.C. Agarwal and J.W. Cooley

Winograd minimum-multiply convolution and DFT algorithms

S. Winograd has proved that a length-N circular or linear convolution or DFT requires less than $2N$ multiplies (for real data), or $4N$ real multiplies for complex data. (This doesn't count multiplies by rational fractions, like 3 or $\frac{1}{N}$ or $\frac{5}{17}$, which can be computed with additions and one overall scaling factor.) Furthermore, Winograd showed how to construct algorithms achieving these counts. Winograd prime-length DFTs and convolutions have the following characteristics:

1. Extremely efficient for small N ($N < 20$)
2. The number of adds becomes **huge** for large N.

Thus Winograd's minimum-multiply FFT's are useful only for small N. They are very important for Prime-Factor Algorithms, which generally use Winograd modules to implement the short-length DFTs. Tables giving the multiplies and adds necessary to compute Winograd FFTs for various lengths can be found in C.S. Burrus (1988). Tables and FORTRAN and TMS32010 programs for these short-length transforms can be found in C.S. Burrus and T.W. Parks (1985). The theory and derivation of these algorithms is quite elegant but requires substantial background in number theory and abstract algebra. Fortunately for the practitioner, all of the short algorithms one is likely to need have already been derived and can simply be looked up without mastering the details of their derivation.

Winograd Fourier Transform Algorithm (WFTA)

The Winograd Fourier Transform Algorithm (WFTA) is a technique that recombines the short Winograd modules in a prime-factor FFT into a composite-N structure with fewer multiplies but more adds. While theoretically interesting, WFTAs are complicated and different for every length, and on modern processors with hardware multipliers the trade of multiplies for many more adds is very rarely useful in practice today.

Efficient FFT Algorithm and Programming Tricks

The use of FFT algorithms such as the radix-2 decimation-in-time or decimation-in-frequency methods result in tremendous savings in computations when computing the discrete Fourier transform. While most of the speed-up of FFTs comes from this, careful implementation can provide additional savings ranging from a few percent to several-fold increases in program speed.

Precompute twiddle factors

The twiddle factor, or $W_N^k = e^{-\left(i\frac{2\pi k}{N}\right)}$, terms that multiply the intermediate data in the FFT algorithms consist of cosines and sines that each take the equivalent of several multiplies to compute. However, at most N unique twiddle factors can appear in any FFT or DFT algorithm. (For example, in the radix-2 decimation-in-time FFT, only $\frac{N}{2}$ twiddle factors $\forall k, k = \left\{0, 1, 2, \ldots, \frac{N}{2} - 1\right\} : \left({W_N}^k\right)$ are used.) These twiddle factors can be precomputed once and stored in an array in computer memory, and accessed in the FFT algorithm by **table lookup**. This simple technique yields very substantial savings and is almost always used in practice.

Compiler-friendly programming

On most computers, only some of the total computation time of an FFT is spent performing the FFT butterfly computations; determining indices, loading and storing data, computing loop parameters and other operations consume the majority of cycles. Careful programming that allows the compiler to generate efficient code can make a several-fold improvement in the run-time of an FFT. The best choice of radix in terms of program speed may depend more on characteristics of the hardware (such as the number of CPU registers) or compiler than on the exact number of computations. Very often the manufacturer's library codes are carefully crafted by experts who know intimately both the hardware and compiler architecture and how to get the most performance out of them, so use of well-written FFT libraries is generally recommended. Certain freely available programs and libraries are also very good. Perhaps the best current general-purpose library is the

FFTW package; information can be found at http://www.fftw.org. A paper by Frigo and Johnson describes many of the key issues in developing compiler-friendly code.

Program in assembly language

While compilers continue to improve, FFT programs written directly in the assembly language of a specific machine are often several times faster than the best compiled code. This is particularly true for DSP microprocessors, which have special instructions for accelerating FFTs that compilers don't use. (I have myself seen differences of up to 26 to 1 in favor of assembly!) Very often, FFTs in the manufacturer's or high-performance third-party libraries are hand-coded in assembly. For DSP microprocessors, the codes developed by Meyer, Schuessler, and Schwarz are perhaps the best ever developed; while the particular processors are now obsolete, the techniques remain equally relevant today. Most DSP processors provide special instructions and a hardware design favoring the radix-2 decimation-in-time algorithm, which is thus generally fastest on these machines.

Special hardware

Some processors have special hardware accelerators or co-processors specifically designed to accelerate FFT computations. For example, AMI Semiconductor's Toccata ultra-low-power DSP microprocessor family, which is widely used in digital hearing aids, have on-chip FFT accelerators; it is always faster and more power-efficient to use such accelerators and whatever radix they prefer.

In a surprising number of applications, almost all of the computations are FFTs. A number of special-purpose chips are designed to specifically compute FFTs, and are used in specialized high-performance applications such as radar systems. Other systems, such as OFDM-based communications receivers, have special FFT hardware built into the digital receiver circuit. Such hardware can run many times faster, with much less power consumption, than FFT programs on general-purpose processors.

Effective memory management

Cache misses or excessive data movement between registers and memory can greatly slow down an FFT computation. Efficient programs such as the FFTW package are carefully designed to minimize these inefficiences. In-place algorithms reuse the data memory throughout the transform, which can reduce cache misses for longer lengths.

Real-valued FFTs

FFTs of real-valued signals require only half as many computations as with complex-valued data. There are several methods for reducing the computation, which are described in more detail in Sorensen et al.

1. Use DFT symmetry properties to do two real-valued DFTs at once with one FFT program
2. Perform one stage of the radix-2 decimation-in-time decomposition and compute the two length-$\frac{N}{2}$ DFTs using the above approach.
3. Use a direct real-valued FFT algorithm; see H.V. Sorensen et.al.

Special cases

Occasionally only certain DFT frequencies are needed, the input signal values are mostly zero, the signal is real-valued (as discussed above), or other special conditions exist for which faster algorithms can be developed. Sorensen and Burrus describe slightly faster algorithms for pruned or zero-padded data. Goertzel's algorithm is useful when only a few DFT outputs are needed. The running FFT can be faster when DFTs of highly overlapped blocks of data are needed, as in a spectrogram.

Higher-radix algorithms

Higher-radix algorithms, such as the radix-4, radix-8, or split-radix FFTs, require fewer computations and can produce modest but worthwhile savings. Even the split-radix FFT reduces the multiplications by only 33% and the additions by a much lesser amount relative to the radix-2 FFTs;

significant improvements in program speed are often due to implicit loop-unrolling or other compiler benefits than from the computational reduction itself!

Fast bit-reversal

Bit-reversing the input or output data can consume several percent of the total run-time of an FFT program. Several fast bit-reversal algorithms have been developed that can reduce this to two percent or less, including the method published by D.M.W. Evans.

Trade additions for multiplications

When FFTs first became widely used, hardware multipliers were relatively rare on digital computers, and multiplications generally required many more cycles than additions. Methods to reduce multiplications, even at the expense of a substantial increase in additions, were often beneficial. The prime factor algorithms and the Winograd Fourier transform algorithms, which required fewer multiplies and considerably more additions than the power-of-two-length algorithms, were developed during this period. Current processors generally have high-speed pipelined hardware multipliers, so trading multiplies for additions is often no longer beneficial. In particular, most machines now support single-cycle multiply-accumulate (MAC) operations, so balancing the number of multiplies and adds and combining them into single-cycle MACs generally results in the fastest code. Thus, the prime-factor and Winograd FFTs are rarely used today unless the application requires FFTs of a specific length.

It is possible to implement a complex multiply with 3 real multiplies and 5 real adds rather than the usual 4 real multiplies and 2 real adds:

$$(C+iS)\,(X+iY)=CX-SY+i\,(CY+SX)$$

but alernatively

$$Z=C\,(X-Y)$$

$$D = C + S$$

$$E = C - S$$

$$CX - SY = EY + Z$$

$$CY + SX = DX - Z$$

In an FFT, D and E come entirely from the twiddle factors, so they can be precomputed and stored in a look-up table. This reduces the cost of the complex twiddle-factor multiply to 3 real multiplies and 3 real adds, or one less and one more, respectively, than the conventional 4/2 computation.

Special butterflies

Certain twiddle factors, namely $W_N^0 = 1$, $W_N^{\frac{N}{2}}$, $W_N^{\frac{N}{4}}$, $W_N^{\frac{N}{8}}$, $W_N^{\frac{3N}{8}}$, etc., can be implemented with no additional operations, or with fewer real operations than a general complex multiply. Programs that specially implement such butterflies in the most efficient manner throughout the algorithm can reduce the computational cost by up to several N multiplies and additions in a length-N FFT.

Practical Perspective

When optimizing FFTs for speed, it can be important to maintain perspective on the benefits that can be expected from any given optimization. The following list categorizes the various techniques by potential benefit; these will be somewhat situation- and machine-dependent, but clearly one should begin with the most significant and put the most effort where the pay-off is likely to be largest.

Methods to speed up computation of DFTs

- **Tremendous Savings**

    1. FFT ($\frac{N}{\log_2 N}$ savings)

- **Substantial Savings** (≥ 2)

1. Table lookup of cosine/sine
2. Compiler tricks/good programming
3. Assembly-language programming
4. Special-purpose hardware
5. Real-data FFT for real data (factor of 2)
6. Special cases

- **Minor Savings**

1. radix-4, split-radix (-10% - +30%)
2. special butterflies
3. 3-real-multiplication complex multiply
4. Fast bit-reversal (up to 6%)

Note: On general-purpose machines, computation is only part of the total run time. Address generation, indexing, data shuffling, and memory access take up much or most of the cycles.

Note:A well-written radix-2 program will run much faster than a poorly written split-radix program!

Choosing an FFT length

The most commonly used FFT algorithms **by far** are the power-of-two-length FFT algorithms. The Prime Factor Algorithm (PFA) and Winograd Fourier Transform Algorithm (WFTA) require somewhat fewer multiplies, but the overall difference usually isn't sufficient to warrant the extra difficulty. This is particularly true now that most processors have single-cycle pipelined hardware multipliers, so the total operation count is more relevant. As can be seen from the following table, for similar lengths the split-radix algorithm is comparable in total operations to the Prime Factor Algorithm, and is considerably better than the WFTA, although the PFA and WTFA require fewer multiplications and more additions. Many processors now support single cycle multiply-accumulate (MAC) operations; in the power-of-two algorithms all multiplies can be combined with adds in MACs, so the number of additions is the most relevant indicator of computational cost.

	FFT length	**Multiplies (real)**	**Adds(real)**	**Mults + Adds**
Radix 2	1024	10248	30728	40976
Split Radix	1024	7172	27652	34824
Prime Factor Alg	1008	5804	29100	34904

	FFT length	**Multiplies (real)**	**Adds(real)**	**Mults + Adds**
Winograd FT Alg	1008	3548	34416	37964

Representative FFT Operation Counts

The Winograd Fourier Transform Algorithm is particularly difficult to program and is rarely used in practice. For applications in which the transform length is somewhat arbitrary (such as fast convolution or general spectrum analysis), the length is usually chosen to be a power of two. When a particular length is required (for example, in the USA each carrier has exactly 416 frequency channels in each band in the AMPS cellular telephone standard), a Prime Factor Algorithm for all the relatively prime terms is preferred, with a Common Factor Algorithm for other non-prime lengths. Winograd's short-length modules should be used for the prime-length factors that are not powers of two. The chirp z-transform offers a universal way to compute any length DFT (for example, Matlab reportedly uses this method for lengths other than a power of two), at a few times higher cost than that of a CFA or PFA optimized for that specific length. The chirp z-transform, along with Rader's conversion, assure us that algorithms of $O(N \log N)$ complexity exist for **any** DFT length N.

Selecting a power-of-two-length algorithm

The choice of a power-of-two algorithm may not just depend on computational complexity. The latest extensions of the split-radix algorithm offer the lowest known power-of-two FFT operation counts, but the 10%-30% difference may not make up for other factors such as regularity of structure or data flow, FFT programming tricks, or special hardware features. For example, the decimation-in-time radix-2 FFT is the fastest FFT on Texas Instruments' TMS320C54x DSP microprocessors, because this processor family has special assembly-language instructions that accelerate this particular algorithm. On other hardware, radix-4 algorithms may be more efficient. Some devices, such as AMI Semiconductor's Toccata ultra-low-power DSP microprocessor family, have on-chip FFT accelerators; it is always faster and more power-efficient to use these accelerators and whatever radix they prefer. For fast convolution, the decimation-in-frequency algorithms may be preferred because the bit-reversing can be bypassed; however, most DSP microprocessors provide zero-overhead bit-reversed indexing hardware and prefer decimation-in-time algorithms, so this may not be true for such machines. Good, compiler- or hardware-friendly programming always matters more than modest differences in raw operation counts, so manufacturers' or good third-party FFT libraries are often the best choice. The module FFT programming tricks references some good, free FFT software (including the FFTW package) that is carefully coded to be compiler-friendly; such codes are likely to be considerably faster than codes written by the casual programmer.

Multi-dimensional FFTs

Multi-dimensional FFTs pose additional possibilities and problems. The orthogonality and separability of multi-dimensional DFTs allows them to be efficiently computed by a series of one-dimensional FFTs along each dimension. (For example, a two-dimensional DFT can quickly be computed by performing FFTs of each row of the data matrix followed by FFTs of all columns, or vice-versa.) **Vector-radix FFTs** have been developed with higher efficiency per sample than row-column algorithms. Multi-dimensional datasets, however, are often large and frequently exceed the cache size of the processor, and excessive cache misses may increase the computational time greatly, thus overwhelming any minor complexity reduction from a vector-radix algorithm. Either vector-radix FFTs must be carefully programmed to match the cache limitations of a specific processor, or a row-column approach should be used with matrix transposition in between to ensure data locality for high cache utilization throughout the computation.

Few time or frequency samples

FFT algorithms gain their efficiency through intermediate computations that can be reused to compute many DFT frequency samples at once. Some applications require only a handful of frequency samples to be computed; when that number is of order less than $O(\log N)$, direct computation of those values via Goertzel's algorithm is faster. This has the additional advantage that any frequency, not just the equally-spaced DFT frequency samples, can be selected. Sorensen and Burrus developed algorithms for when most input samples are zero or only a block of DFT frequencies are needed, but the computational cost is of the same order.

Some applications, such as time-frequency analysis via the short-time Fourier transform or spectrogram, require DFTs of overlapped blocks of discrete-time samples. When the step-size between blocks is less than $O(\log N)$, the running FFT will be most efficient. (Note that any window must be applied via frequency-domain convolution, which is quite efficient for sinusoidal windows such as the Hamming window.) For step-sizes of $O(\log N)$ or greater, computation of the DFT of each successive block via an FFT is faster.